Cornelia Boldyreff
Design Frameworks - a Basis for Conceptual Understanding and Reuse
Abstract

Software reuse is often advocated as a way of increasing productivity, quality and maintainability. This thesis advocates the view that reuse of software structuring concepts will allow reuse to take place at a much earlier stage in the software lifecycle, the architectural design stage; and that reuse based on higher level abstractions than simply code will lead to improved software design. However, reuse of such design concepts requires an appropriate way of understanding existing software designs, representing design concepts in such a way as facilitate reuse, and then encouraging concept reuse on new projects. Here, this thesis puts forward a way of developing structured concept descriptions based on a concept description form (CDF). This is the main contribution of this research. The CDF is determined by analysing the requirements that reuse of design concepts places on such a form. The CDF has been developed to meet these requirements. Briefly the CDF allows existing software systems to be described in terms of their underlying concepts with provisions for describing design concepts at various levels of abstraction.

The CDF is then applied to a small number of examples, compared to other related developments, and finally compared with a more conventional approach to representing software for reuse, faceted classification. As an illustration of usage, the CDF is applied in a small scale study of software concepts used in compilers. As a more substantial test bench, experiments are then undertaken applying the CDF in an industrial application domain, that of steel production.

The main results of this work have been to establish the CDF as a viable form for describing reusable software concepts; and to show that the CDF is capable of being used in an industrial application to support software concept reuse. It is concluded that the CDF provides a means of recording the understanding of design concepts at various levels of abstraction and thus provides a basis for their reuse in future designs.

Design Frameworks - a Basis for Conceptual Understanding and Reuse

Cornelia Boldyreff

Ph. D. Thesis

University of Durham
School of Engineering and Computer Science
Computer Science

June 17, 1994

Acknowledgements

Many people have helped me with the work reported here. I would like to thank Professor Patrick A. V. Hall, my original supervisor. Since joining the School of Engineering and Computer Science at the University of Durham, I have received much help from my colleagues; I would especially like to thank Malcolm Munro, my supervisor, and Professor Keith Bennett, my advisor. I owe a deep debt of gratitude to Professor Dr.-Ing. Peter F. Elzer; without his support and encouragement, I would never have been able to undertake the software field studies which enabled me to bring this work to fruition.

Many friends and members of my family have given me encouragement to complete this thesis; and they know how grateful I am. I would like to dedicate this work to the memory of my father, Dr. Ephraim Basil Boldyreff.

Declaration

The material contained in this thesis has not been previously submitted for a degree in this or any other university.

Much of the work described in this thesis was carried out with the support of the CEC under the ESPRIT programme on Project 1094, Practitioner. The collaborators in this project were Asea Brown Boveri AG (ABB), Computer Resources International, PCS Computer Systeme GmbH, Brunel University, the Technical University Clausthal (TUC), and the University of Liverpool. Professor Dr.-Ing. Peter Elzer, formerly at ABB, now at TUC, and Dr. Jan Witt at PCS were responsible for the original ideas which formed the starting point of the research within the Practitioner Project. While the author did not participate in the development of the original questionnaire for describing reusable software concepts, its development and refinement through application provided the main research focus of the author's work in the project leading to the development of the concept of design frameworks by the author. Unless otherwise stated, all the research work carried out in support of this thesis has been the responsibility of the author.

Statement of Copyright

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter sets the context for this research within the field of Computer Science known as Software Engineering and, in particular, within the area of Software Reuse. Research directions and outstanding issues are summarized. The need for work on improved means of software description to support reuse is discussed; and the specific concerns to be addressed by this thesis are outlined.

1.1 Perspective

The distinction between Computer Science and Software Engineering, based on the distinction between any science and engineering, is that of theory versus its application in practice. But it must not be forgotten that engineering is not *just* applied science; it is in its own right a creative and innovative undertaking; see, for example, McDermid's remarks in [99] made in the context of developing a definition of software engineering. As the engineer, Petroski, has remarked the principal object of engineering is the world that engineers themselves create, not the given world of nature [116]. Taking Computer Science as the study of the theories of computational machines and computational functions, and the mathematical modelling associated

with those machines and functions (an amplification of Brady's characterisation found in [38]), Software Engineering can be taken as the creative and innovative application of theories and models of computational machines and functions in the development of software systems along with the theory of other sciences, both natural and artificial (in the sense of [144]), depending on the intended application of the developed systems. The point that science and engineering are linked in developing computer applications was made as early as 1966 [107] by the then president of the Association of Computer Machinery:

> *A concern with the SCIENCE of computing and information processing, while undeniably of the utmost importance and an historic root of our organization (i.e. the ACM) is, alone, too exclusive. While much of what we do is or has its root in not only computer and information science, but also many older and better defined sciences, even more is not at all scientific but of a professional and engineering nature. We must recognize ourselves - not necessarily all of us and not necessarily any one of us all the time - as members of an ENGINEERING profession, be it hardware engineering or software engineering, a profession without artificial and irrelevant boundaries like that between "scientific" and "business" applications.*

The recognition that the systems designer is not simply an applier of scientific theory, as postulated by the traditional application paradigm, also comes from insights into the design process that can be found in the account of the part that theory plays in design given by Greif [70]. Theories are artificial tools for thought from this viewpoint, as Greif explains in the following quotation:

> *Even if they are complex, scientific theories, methods and experiments simplify practical problem conditions. It would be therefore naive if the designer would trust too much in theories and theoretical design concepts if he wants to solve practical problems. He should know - and this is an application problem shared by all social, natural, and technical sciences -*

that theories are idealized representations of reality or artificial tools for thought. ...

In other words theories are frameworks that systematically guide processes of problem solving. The practical success of the designer therefore depends not only on the quality of the theory but also on his knowledge of the task and application situation, together with his theoretical background knowledge, theoretical creativity, and - last but not least - his practical and social competences in marketing the artifact.

The thesis which follows gives a concrete interpretation of this rather abstract view of design with respect to software design through the development and application of a means of structuring descriptions of software concepts to support their reuse. The notions of a software concept and an associated design framework will be elaborated more fully in the body of this thesis. This thesis is written from the perspective of a software engineer and its central concern is software reuse. This is relatively new research area in software engineering although its importance has been recognised increasingly over the last ten years.

1.1.1 A Short History of Software Reuse Research and Associated Publications

In the report of the Alvey Committee, a vision of an information system factory can be found [10]. One of the factory's five main subsystems is envisaged to be *a database or knowledge base of available software and hardware components.* Already the need to reuse existing software and hardware components in order to compete effectively in producing IT application systems was appreciated. The Alvey Committee believed that in the medium term, from five to ten years (i.e. by 1992 at the upper limit), that the nature of system development would be changed by the *development of reusable software and hardware modules, rigorously tested and formally documented.* Their report highlighted the need for a shift of attention from the source code representation of programs to requirements and designs, and stressed

that projects must be far more concerned with the *higher level* representations. Here the text notes parenthetically that such a shift of attention is entirely compatible with an approach which emphasises the reuse of existing components. The work on reuse within the ECLIPSE project funded under the Alvey programme can be traced to these concerns expressed in the report, particularly the need for methods of structuring software components for ease of configuration, addressed in part within ECLIPSE by the development of SySL (System Structure Language) on which the research for this thesis draws.

The Alvey committee's vision of software component brokerage unfortunately has yet to be realised; the practical problems of organising software reuse on a large scale are still not sufficiently well understood for its widespread practice in this way to be commercially viable. This thesis will not address these problems.

In the year following the publication of the Alvey report, the ITT workshop on reuse brought together many of pioneering researchers in the field of software reuse in the USA. Here Freeman established many of the basic concepts of reusability with his proposed programme of reusability research [64]. He defined a hierarchy of objects of reusability, i.e. any information needed by a developer in the process of creating software. His hierarchy consists of five levels: environmental (technology transfer and utilization knowledge), external (development and application area knowledge), functional architectures (generic systems and functional collections), logical structures (software architectures) and code fragments. As Freeman pointed out, reusable information is not qualitatively different from that found in well-documented software engineering work products; and he argued that this implies that the core of the reusability problem is to make sure that systems are properly represented. Many of the key papers from this workshop were reproduced in the 1984 IEEE Transactions on Software Engineering special issue devoted to Software Reusability [18] and in Freeman's IEEE Tutorial on Software Reusability [81]. Sufficient research was undertaken in this period for another IEEE Tutorial on Software Reuse by Tracz to appear in the following year [82]. The Tracz tutorial contains a reprint of a report of U.K. workshop on software reuse held in 1986 where work in progress under the Alvey and other initiatives was discussed [132]. At this time, reuse research in the

U.K. was divided between those advocating formally specified components and those focusing on organisational issues related to reuse. Much of the work reported was in progress.

In 1989, the ACM Press published two volumes on Software Reusability edited by Ted J. Bickerstaff and Alan J. Perlis [2]. These two volumes bring together several of the key papers on software reusability that have been written during the eighties. Many of these papers have already appeared in the lesser known volume, ITT Proceedings of the Workshop on Reusability in Programming (1983), in the IEEE Software Reusability Tutorial edited by Freeman (1986) and other IEEE publications such as IEEE Transactions on Software Engineering and IEEE Software. In the first volume, there can be found a framework for reusability technologies, discussed in Chapter 2, and the conclusion with respect to future directions that the fundamental problem preventing the successful reuse of design information is finding the right representation of that design information [20].

In bringing together so much existing material, the two editors made some efforts to concentrate on the Very Large Scale Reuse (VLSR) Problem. Although they contrast this with the narrow code-oriented viewpoint, they do not go far enough in characterising the need for appropriate abstractions as the key issue in making VLSR possible. The principal merit of these volumes is the division of the papers into the two broad classes of theory and practice; and it is notable that theory has been put before practice although an astute reader will notice that the dates at which the original papers were written do not necessarily reflect this ordering. It is clear from these volumes that both the theory and practice of reuse of software are in their infancy. Nevertheless, these two volumes form a good introduction for any software engineer wishing to become acquainted with the research and practical applications in the field of software reuse.

A very notable omission from both the ACM and IEEE publications is material describing European research and practice of software reuse; this has been recently addressed by the following publications [72, 56, 159].

The research in this thesis builds on the foundations of existing research in reuse with its established theme of representational issues affecting reuse. The research in this thesis is concerned with the representation of software design concepts in order to facilitate their reuse and with the processes of designing for reuse and designing with reuse. As far back as the report of the Alvey committee the representational aspects of reuse have been recognised as requiring attention. This thesis seeks to develop a new perspective on the role of reuse in software development and argues that reuse is a fundamental part of design. A broader perspective of software engineering research now views the whole of the software life cycle as inherently encompassing both software reuse and maintenance [51]. A long-term action identified as part of this agenda is the building of a unifying model for software system development. The considerations of how best to design for reuse and with reuse found in this thesis are contributions to the groundwork required for such a unified model.

1.1.2 Influences from General Engineering

Software development as an engineering process can usefully draw upon the results of studies in Design Theory from general engineering research [106]. The research reported here has done so; see particularly chapter 5 of this thesis. From an engineering perspective, the activity of design is itself amenable to be being designed. Thus, the potential exists to develop theories of design with the possibility of radically changing the nature of existing design practice.

Moreover it is important to understand that engineering method is not the same as scientific method. For a far ranging discussion of the differences between scientific method and the engineering design method, see [58]. In considering how software design is to proceed in order to support the reuse of software concepts explicitly, one goal of this research is an improved understanding of design approaches with respect to the processes and the products of software engineering.

It is increasingly recognised that software engineering is an engineering discipline concerned with the development of **large** software systems, for example, see Som-

merville's Preface to [147]. These large system developments could not be accomplished without the co-operative working of several software engineers. With the advent of Computer Aided Engineering, many process models of development with associated method and tool support have emerged [110, 39, 101], but comparable models, methods and tools for large scale software engineering design are lacking [52, 143]. As McDermid has pointed out, most improvements in our technology for large scale software system development have depended upon the finding of improved abstractions or structuring techniques for describing software [99]. He identifies structuring and abstraction as the two key weapons for mastering the difficulties of building large scale software systems. Looking for improved abstractions and appropriate structuring techniques to support software concept reuse has provided a starting point for the research reported here.

1.2 Contribution to the Subject

The key issue addressed by this research in software reusability is software concept description within design frameworks. The notion of a software concept used in this thesis is derived from that developed within the Practitioner project which will be discussed in Chapter 2. For now, the reader may rely on their everyday understanding of the term concept applied to software. For example, in [35], the first part is entitled *Basic Concepts*. Here Bornat introduces software structuring concepts such as sequences of instructions and procedures and anticipates the discussion on structured instructions such as repetition and choice. These are fundamental software concepts. In a particular software application area, more complex software concepts have been described; for example, in financial applications, the software concept of a spreadsheet is a recently developed concept. Of course, these concepts, such as repetition and choice or a spreadsheet, have a usage that extends beyond their realisation in software; but it is by virtue of the fact that they can be realised in software that they are spoken of as software concepts in this context. The notion of a design framework will also be developed in subsequent chapters. In this context, a

design framework is an abstraction over known design structures which in the case of software will be the structuring concepts found in software designs. This issue of software concept description within design frameworks has been tackled through the development of a means for the recording and relating software concepts within design frameworks. It will be shown how design frameworks provide a means of describing generic architecture for classes of applications and a vehicle for reusing architectural principles and guidelines in the development of new systems. Briefly put they promise a partial solution to the following problem stated in [21]:

> *The engineering of large systems by interconnecting reusable software components from diverse sources in a systems integration activity is problematic. Although the technology for large systems integration exists (viz applications frameworks and applications generators), there are few well developed methods which address the engineering of large systems.*

The approach taken in this research has been to investigate means of capturing the architectural structures of system models used in system design by studying typical examples and by experimenting with various descriptive forms. Standard frameworks (e.g. the multi-phase compiler model, the OSI reference model, the Purdue University model of hierarchical control systems in steel production) relating various system designs in specific domains have been investigated. The research has concentrated on system models employed in the domain of steel production. A particular interest has been to determine how frameworks can be abstracted from known systems models, and to determine the role that levels of abstraction within a framework play in design.

Currently research in software reuse is focused on the reuse of software components - what might be characterised as "reuse-in-the-small"; the contribution of this research is to show how "reuse-in-the-large" can be supported by providing an approach which looks beyond components to frameworks within which architectural designs relating components can be reused. This approach far from replacing component composition allows it to proceed in a more appropriate context.

1.2.1 Genesis of the Research and its Goal

Very early on in the work, an interest developed in what can be termed the composition problem. When reuse takes place in the context of a library of software components, how is the designer to effectively create a new system from an assortment of retrieved components? One approach which originally seemed obvious was that this was simply an interfacing problem. However, in design terms, the introduction of interface transforming components seems inelegant, and in practical terms this approach is likely to give rise to inefficient systems encumbered by additional interface transforming components. The strongest objection to this approach is that systems so composed are difficult to understand and reason about because they are lacking a conceptual unity. Rather than developing the approach of generating systems from components composed with the software equivalent of "glue", a study has been undertaken of the process of design with a particular concern for how designers apply decompositional approaches to design and how the reuse of existing decompositions can be supported. In this area, the work of Alexander, especially his **Notes on the Synthesis of Form** [9], and the work of Jones (**DESIGN METHODS seeds of human futures**) [85] have been found to be particularly relevant as both give characterisations of the design process which are discussed in greater detail in Chapter 6.

As the research is concerned with the reuse of software concepts as realised in existing software, a study has been made in parallel of how best to describe software concepts. In the context of developing guidelines for the description of software concepts, the problem arose of how to decompose existing systems into their immediate parts for the purpose of description prior to entering these software concepts into a library. A canonical form of software concept description has been developed; this descriptive form is simply used recursively to describe any immediate parts until the descriptions pertain to atomic software concepts, but use of the form to describe software has been problematic.

The research brought together software process concerns with software product concerns when an attempt was made to give more thought to how a reuse support

system would be used in practice by designers. The importance of structuring a design problem so that it could be attacked at various levels of abstraction was evident in the accounts of design theory that were studied. Here it was found that the use of reference models employed in standards development and the looser concept of frameworks could be generalised and applied to systems development. Further confirmation of this approach was provided by acquiring the reports from Purdue University which describe Steel Plant control systems using a four level model. The availability of such a model proved to be of great assistance when the task of describing software concepts from existing systems documentation of applications from the Steel Production domain was tackled. Experience in the Steel Production domain has shown that where systems have been designed in the framework of a common system model, the model itself provides insight into understanding the software concepts and their interfaces that are likely to found on examination of the existing software. This model is especially helpful when attempting to identify the concept's interfaces; particularly where the description of the concept is restricted to a particular level or levels of the model. Further work on this model has formed the practical application of concept description form developed as part of this research.

The subtext of this thesis is the all pervasive nature of design concerns in reuse of software concepts - be they architectural concepts or component concepts - in systems engineering. So although studying how design concepts can be derived from existing system models, and subsequently reused in the design of new systems, an attempt has been made where possible to relate this research within the wider context of existing theories of design.

It is not the intention of this research to develop a new theory of design; the aim is simply to improve understanding of the design process in the context of software engineering. The work has been confined to examining what is useful to apply towards solving the specific design problems of software engineering. Some consideration has been given to accounts of the design process in general; and having identified three essential aspects of design - cognition, conceptualisation and construction, the research has been concentrated within the conceptualisation phase.

1.3 Aims and Plan of Thesis

In the Chapter 2, a more detailed survey of reuse research and the specific research work undertaken in an ESPRIT project, the Practitioner project, is made; and the research supporting this thesis is related to wider research concerns in domain analysis and software reusability which can be characterised as design-for-reuse and design-with-reuse respectively. This thesis develops an approach to reuse that addresses both design-for-reuse and design-with-reuse. The nature of design is such that the two objectives of design:

- understanding and recording what has already been accomplished, i.e. existing design concepts, and
- understanding and recording what is to be developed building on existing conceptual foundations, i.e. new design concepts

can be interpreted as evidence that implicitly all design involves reuse. This thesis aims to provide a means whereby such implicit reuse can be made more explicit.

In Chapter 3, work on the representation of software concepts in the context of identified reuse requirements is given consideration. The ability to utilize abstract representations is crucial to design [104] and software reuse [64, 93] *et al.* There is a need for investigations into the appropriate levels of abstraction in particular fields of design discourse [64]. This chapter establishes the adequacy of a canonical form for describing reusable software concepts - the concept description form developed within the ESPRIT Practitioner Project by the author. This form is then used in Chapters 4 and 5 to describe software concepts abstracted from existing software systems through a process of domain analysis including investigative software field studies.

Chapter 4 describes a small scale application of the concept description form. Here the form is used to describe design concepts used in the construction of language processors. Chapter 5 gives results of a larger scale application in an industrial setting.

While the concept description form has the potential to support design generally in other fields of engineering, the reported large scale application is limited to software engineering design in the area of metallurgy (chosen because the metallurgy business division of the industrial partner taking the role of internal customer on the project where this research was carried out volunteered to participate). Here the application of the research to give some realistic content to the work has been an examination of software system concepts within a particular application domain of metallurgy, that of process control in steel production.

Chapter 6 evaluates these applications and presents a further development of the work on concept description forms whereby sets of forms describing related software concepts can be interpreted as constituting a design framework. Such design frameworks provide a means for expressing the results of software investigations in a structured form to facilitate their understanding and subsequent reuse in new developments. The role that concepts play in structuring design has been a key concern for this investigation. Building on insights into the role that language plays in facilitating understanding and recording of designs and their communication, and the thesis (of Frege, see Chapter 2) that only in the context of a proposition does a word refer, this thesis argues that software concepts can only be understood in context. The concept description form developed and applied within these chapters provides a means of recording the contextual understanding of designs, so that the reuse of software concepts in design may be supported explicitly.

Chapter 7 discusses the contribution of this work in the field of Software Engineering and the broader implications of this research in the context of research on Design Theory. It indicates directions for future research; and it concludes the thesis evaluating how and to what extent its overall aims have been fulfilled.

1.3.1 Research Method and Criteria for Success

The research method employed in this study has been derived from the following design strategy: understand the requirements of the problem, develop and evaluate a

solution with respect to the original requirements and its application in practice. In this case, the main problem tackled has been to develop an adequate representational form for software concept descriptions that supports their reuse and to apply this form in domain analysis to support software concept reuse in practice. The form developed is seen primarily as a means of supporting software concept reuse in design, both in design-for-reuse and design-with-reuse. In both the development and application of the form, an analysis of requirements has been made and a solution proposed and demonstrated.

The representational form has been employed in the description of sets of software concepts which have subsequently been used to constitute design frameworks. The design framework developed here has been applied in the domain of steel production in a series of studies. These studies have been an essential part of the research in order to validate and refine the design framework itself. Thus this thesis is an application of Software Engineering, and while methods from Computer Science have been applied in the work described, they are not a primary concern for evaluation.

This research on software concept reuse supported by practical application has required significant research on the software concepts and state of design practice in the domain of Steel Production.

The expected results of this research arise from four main aims:

- to develop further a generic form for describing software concepts;
- to establish the adequacy of the representational form for describing reusable software concepts;
- to employ the form in small and large scale applications and evaluate the form as a means of supporting software concept reuse through populating a design concept database;
- to gain an understanding of the inherent role that concept reuse plays in design and to make this more explicit through providing a means of recording design frameworks.

The first two aims will succeed if the arguments in Chapter 3 are accepted. The third aim is based on the case made in Chapter 4 of this thesis, supported by a more thorough going application in the development of a design framework and analysis of its usage in practice found in Chapter 5. The final aim is dependent on the comprehensibility of this thesis taken as a whole.

In developing a representational form to support software concept reuse, a rationale has been developed which relates the representational form to identified requirements, particularly, the needs for abstraction and structuring in design. Validation of this approach to support software concept reuse has been achieved in two ways: one through application by the author in a series of case studies and subsequent demonstrations based on these, and two through consultation with domain experts. Consultation with domain experts has been especially important in order to validate the results of the case studies as the author had little knowledge of metallurgy and steel production before undertaking this research.

The originality of this research lies in its identification of the implicit nature of reuse in design and its development of a concept description form to make such reuse in design more explicit. The approach of employing high level concept descriptions in design is not original; however, the approach developed and demonstrated here with respect to supporting software concept reuse in design is original. Moreover, before this work was undertaken, published accounts of software concept reuse in the domain studied were not available although as discussed in Chapter 5 the need for these was recognised. So the domain studies carried out to support this thesis constitute an original contribution to the application of software engineering in this domain.

1.4 The Relationship Between this Research and the Practitioner Project

Much of the work described in this thesis was carried out with the support of the CEC under the ESPRIT programme on Project 1094, Practitioner. The collaborators in this project were Asea Brown Boveri AG (ABB), Computer Resources International, PCS Computer Systeme GmbH, Brunel University, the Technical University Clausthal (TUC), and the University of Liverpool. Professor Dr.-Ing. Peter Elzer, formerly at ABB, now at TUC, and Dr. Jan Witt at PCS were responsible for the original ideas which formed the starting point of the research within the Practitioner Project. While the author did not participate in the development of the original questionnaire for describing reusable software concepts, its development and refinement through application provided the main research focus of the author's work in the project leading to the development of the reusable design frameworks by the author. Unless otherwise stated, all the research work carried out in support of this thesis has been the responsibility of the author.

Chapter 2

Approaches to Software Reuse: Design-for-Reuse and Design-with-Reuse

The aim of this chapter is to provide a more detailed discussion of the background and scope of the work, both from the standpoint of general research in software reuse and the specific research projects with a concentration on the Practitioner Project, in order to set the context within which the research supporting this thesis was undertaken.

2.1 Background to Software Reuse Research

In this section, the topic of reuse is discussed under three main headings in an attempt to answer the following three questions:

- What is reuse?

- Why reuse software? and
- How is it accomplished?

The concept of reuse is explored generally; and in the context of software engineering, the reuse of software concepts is examined as the most general case of reuse. The potential benefits of reuse are expounded. An abstract model of the reuse process with respect to software is elaborated, and various approaches to assisted reuse of software are described.

2.1.1 The Nature of Reuse

The history of ideas, intellectual progress, provides a paradigm for reuse. Here progress is effected by developing and refining the ideas of others. If "learning to speak is learning to think" i.e. language is a precondition for thought, then it is language which underlies our intellectual progress individually as well as collectively; or as the developmentalists say: "Ontogeny recapitulates phylogeny." Generally an armoury of ideas which allows problems to be attacked and solved underlies progress in a particular field. A less combative approach to reuse might describe this armoury of ideas as a treasure house implying that ideas have an intrinsic value. The armoury of ideas can be divided into theory and praxis: agreed principles and accepted practice.

In the field of Software Engineering, the theory largely derives from the application of mathematics, particularly logic; whilst the practice is based on application of engineering methods. On the bookshelf of a software engineer can be found titles such as: *Programming from First Principles*; *Principles of Program Design*; *The Science of Programming*; *Principles of Programming Languages*; *Principles of Compiler Design*; *Compilers Principles, Techniques, and Tools*; *The Theory of Computer Science*; *Fundamental Structures of Computer Science* along with *A Practical Handbook for Software Development*; *The Art of Computer Programming*; *Software Tools*; *Specification Case Studies*; *Numerical Recipes in C*; *Strategies for Real-Time Speci-*

fication and so on. Thus the software engineer combines both formal and informal methods and established principles with proven practice and custom.

The value of studying existing software has been well established amongst software engineers with the publication of algorithms and annotated software sources. This literature forms a rich source of software concepts. Knuth has argued persuasively that better understanding of programs can be achieved by considering programs as *works of literature* [92]; he proposes adoption of a *literate programming* approach to software construction as follows:

> *The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that nicely reinforce each other.*

Note his linkage of comprehensibility with the order and manner of introduction of the underlying program concepts.

Literature is one medium for passing on concepts from one generation to the next; it is part of our cultural heritage. Program literature as proposed by Knuth ensures software concepts are available for reuse from one generation of programmers to the next; however, current practice falls short of this ideal. Developers often lack the time and training to explicate reflectively the conceptual foundations of the systems they develop.

Over the years, however, some consensus concerning the building blocks of software has been achieved eliminating the need to start each development anew. Reuse of common code sequences through macros and the generalised high level language constructs such as structured statements e.g. conditional statements and repetitive statements together with the subroutine library concept have provided the engi-

neering foundations of software construction. And our collective understanding of the behaviour of programming language statements has also contributed to these foundations. Denvir has made this point as follows [53]:

> *The equations and rules governing the behaviour of program language statements are the 'Ohm's and Kirchoff's laws' of programming; just as Ohm's and Kirchoff's laws do not hold for non-metallic conductors or conductors mounted on imperfect insulators, so our rules of program statements do not apply if we use materials outside the collection whose behaviour we have defined.*

The standardisation of programming languages, graphics, database and communications software is an indicator of some maturity of these areas of software technology.

The widespread employment of software packages provides further evidence of software reuse although standardisation is lacking amongst packages. While much study has been devoted to formulating the design principles of system software: operating systems, language processors, editors, database systems, etc, most major applications software subfields lack descriptions of their general design principles and construction methods. As Jones has remarked in [86]:

> *The programming community, therefore, finds itself constantly faced with reinventing concepts which should be available from standard references, but in 1984 are not.*

The situation has not altered much in the intervening years. In order to describe general design concepts, it is necessary to abstract these from studying a number of systems. Shaw has outlined the case for such higher level abstractions in software engineering [140].

The key to reusability of software is abstraction (see [93, 64, 168, 150] and others). The potential reuser of software must be able to find a connection between what

is already known and what is known to be required, between theories and concepts realised in existing software and those comprising a solution to the requirements, by abstracting away unnecessary detail. This paradigm applies not only to software development, but to problem solving generally. The classic reference is Polya's handbook on heuristics: **How To Solve It** [118].

Taking the view that concept formation underlies understanding, three important principles facilitate the understanding of software systems: decomposition, unification and abstraction. Decomposition is the separation of a complex system into its elements or simpler constituents. Unification is the process of organising discrete elements of systems into groups; it forms the basis of later generalisations. Abstraction is the process of forming concepts which allow general classes of elements to be described independent of any system in which they occur. Here a free translation of Vygotsky's account of the process of concept formation [164] has been made relating it to software systems understanding.

A recognised advantage of abstraction in the software life cycle is to separate the specification from any particular implementation. For the purposes of reuse, the specification gives a clear statement of the theory and concepts underlying the software separated from any implementation details; and makes possible their re-employment in the specification of other software. The specification forms a descriptive theory of the software allowing reasoning about the properties and behaviours of the software which may be relevant for reuse.

2.1.2 Motivations for Reuse of Software

The usual reasons put forward for the reuse of software are often economic, such as increased productivity, and to do with improving software quality, such as increased reliability of software, for example, see Lubars quoted below. While these reasons are undoubted important, they are best addressed by getting to the heart of the matter and improving the software development process itself by formulating a more systematic approach to design in which design structuring techniques explicitly

supporting reuse are an important element. Below various requirements are listed with supporting texts to give an indication of why software reuse holds out some promise and why this promise provides a motivation for the development of a reuse based design structuring technique.

There is a requirement for software development approaches which result in reliable systems. Reuse of well-tested components results in more reliable and economic systems [97]. The cost of developing reliable software components can be amortized over several products if the component is reused. That reusability can satisfy this requirement is supported by the following quotes from Lubars [97]:

> *Software reusability can be used to achieve much higher levels of software reliability by amortizing the debugging costs among the products incorporating the reusable component. This largely depends on developing the component to be reusable from the beginning, and debugging it to the desired level of reliability before releasing it.*

and Hoare [77]:

> *Reliable assembly of prespecified parts is an essential mark of maturity in any engineering discipline.*

There is a requirement for progressive software development approaches which enable us to tackle larger more complex system construction projects by building on our previous efforts. Significant progress will not be possible if all projects continue to start from scratch and carry on reinventing the wheel as the following quote from Burstall and JA Goguen [43] concludes:

> *Another important factor for the practical utilization of abstract specification languages, is to build up a library of specifications which can then be used in putting together other larger specifications. ... Without such a library, every program specification effort will have to start from scratch, and there will be no significant progress.*

There is requirement for more control in the software development process; and this will come from a better understanding of software development in practice. One aspect of this is the need for control of *sharing* found in practice as recognised by Dijkstra [55] in the following quote:

> *What the manager sees as "keeping options open" is seen by the scientist as "sharing": different programs sharing code, different proofs sharing arguments, different theories sharing subtheories, and different problems sharing aspects. The need for such sharing is characteristic of the design of anything big. The control of such "sharing" is at the heart of the problem of "scaling up" and it is the challenge to the computing scientist or mathematician to invent the abstractions that will enable us to exert this control with sufficient precision.*

Dijkstra here is alluding to the problem of scaling up the application of mathematics to computing, e.g. assistance with program transformation and theorem proving.

There is a requirement for established mechanisms for the recording of software product knowledge, its structure, components and functions, and channels for its transmission. This knowledge often based on practical experience gained during the development process that is lost if there are no means for its recording and transmission to other practitioners.

Belady and Lehman distinguish between *process knowledge* and *product knowledge* [17]; between knowledge of methods, techniques and tools appropriate for the process of designing software, and the understanding of the software product: its components, structure, functions both individually and collectively and their interactions.

It is the latter knowledge often based on practical experience gained during the development process that is lost if no established mechanisms and channels exist for its recording and transmission to other practitioners for subsequent reuse. It is this final requirement that the structured approach to concept description developed in this thesis seeks chiefly to address.

2.1.3 Supporting Software Reuse in Practice

Various systems developed to support reuse are reviewed below with respect to their underlying models and methods. To aid the discussion a very general model of the reuse process is given initially.

The Reuse Process Underlying the reuse of software concepts is the process of reuse which addresses both the identification of reusable concepts and their deployment in new applications. Because identification and deployment are likely to take place at different times, it is assumed that concepts are somehow stored and held for retrieval later, for example, in a library or database. The reuse process can be summarized involving six main activities as follows:

1. Recognition,
2. Decomposition/Abstraction,
3. Classification,
4. Selection/Retrieval,
5. Specialisation/Adaption, and
6. Composition/Deployment.

The first three activities serve to build up the reuse concept collection; while the latter three serve to make use of the reusable concepts in design.

Initially the opportunity for software reuse will need to be recognised. This may reflect an accumulation of existing software and a recognition that many similar systems have been developed in a particular application domain. Working from existing software, this may involve the decomposition of large software systems into their component concepts and extraction of specific reusable software concepts. From a number of similar specific software concepts, it may be possible to abstract a

reusable software concept that is generic in some sense, perhaps because it has been parameterised, or perhaps because it has been abstracted a level above the level at which it was originally deployed.

There will potentially be many reusable concepts. These will need to be catalogued and stored for retrieval later as required; the appropriate technology for this will need to be made available. The concept description could form the basis of its storage and retrieval. A common method for describing concepts is the classification of the concepts according to some schema. Approaches to document classification in library and information science, particularly the method of faceted classification, may be used to provide a basis for the classification of textual descriptions of software concepts [123].

In putting together a new application, a search for suitable concepts must be undertaken. Once found, these may require specialisation or adaptation before being composed to form the new application.

This process is depicted by a data flow diagram in Figure 2.1 - The Reuse Process. This general model of the reuse process is based on a fuller analysis of the reuse process and the information involved in reuse which can be found in [72]. This rather abstract view of the reuse process is used below to relate various approaches to supporting reuse as well as to characterise four areas of research covering reuse and design in software development.

Systems Supporting Reuse - Their Underlying Models and Methods

Reuse Support Systems relate to the process of reuse described although some may focus on particular aspects of the reuse process such as recognition and decomposition, description, classification and selection whilst others may focus on the selection, specialisation and composition phases. In so far as a Reuse Support System addresses the complete reuse process, it is similar in form to a Project Support Environment (PSE) that provides facilities to support the full project life cycle from capture of requirements through to operational support and maintenance of the

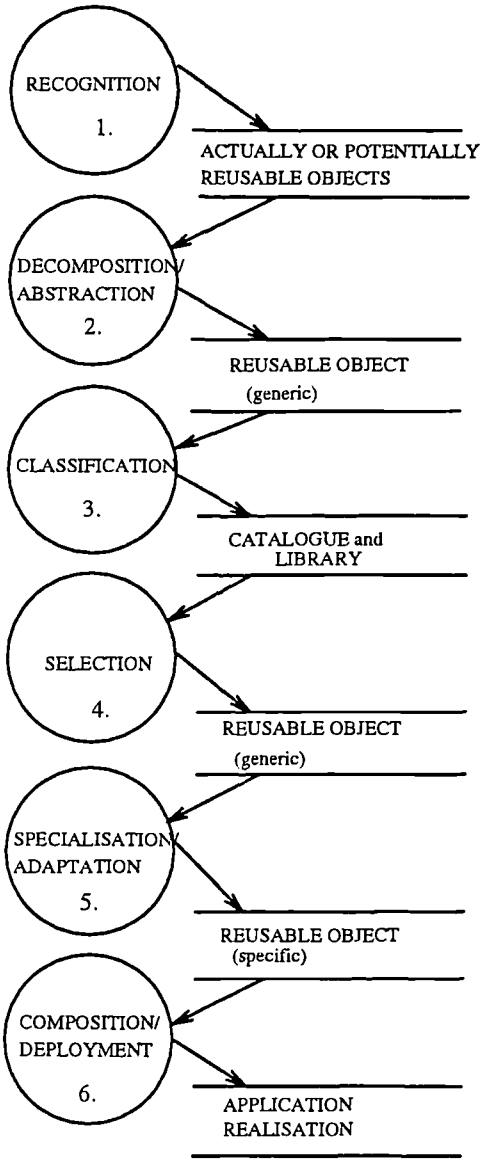


Figure 2.1: The Reuse Process

project software products.

PSEs are often described as integrated where tools have compatible interfaces with the users, other tools and the host environment. In most integrated PSEs (IPSEs), the basis of internal tool integration is an information management system. In a deeper sense of meaning, IPSEs are integrated by embodying an underlying model of the development process which enforces a uniform discipline on the development, a common model of the objects of development and methods to support these models.

Similarly a Reuse Support System may be integrated by its underlying model of the reuse process, the model of the objects of reuse describable in the system, and the methods employed to support these models. At the centre of these support systems, there is a collection, e.g. a library, of reusable objects which provides the basis of internal integration.

A broad distinction has been made between Reuse Support Systems depending whether they are based on the principle of composition or the principle of generation; in their framework for reusability technologies (reproduced in Figure 2.2 - A Framework for Reusability Technologies), Biggerstaff and Richter classify typical systems using these two principles. (This figure has been adapted from [20].) In this framework, typical systems based on composition technologies such as libraries of subroutines, object oriented systems and UNIX pipe architectures are contracted with typical systems based on generation technologies such as Very High Level Languages, Programmer Oriented Languages, CRT Formatter Generators, File Management Generators, and Language Transformers.

Both principles, composition and generation, are important and a Reuse Support System which emphasizes one over the other is incomplete and of limited applicability. While such frameworks can play an important role in helping to classify existing research concerns in Reuse Support System development, there is danger of research into reuse becoming constrained by a false dichotomy such as composition versus generation in this case. This view also restricts itself to component reuse viewing reuse based on generation technologies as execution of component generators.

Features	Approaches to Reusability				
Component Reused	Building Blocks		Patterns		
Nature of Component	Atomic and Immutable Passive		Diffuse and Malleable Active		
Principle of Reuse	Composition		Generation		
Emphasis	Application Component Libraries	Organisation & Composition Principles	Language Based Generators	Application Generators	Transformation Systems
Typical Systems	- Libraries of Subroutines	- Object Oriented - Pipe Architecture	- VHLLs - POLs	- CRT Formatter - File Management	- Language Transformers

Figure 2.2: A Framework for Reusability Technologies

A preferable distinction is between systems that support small-scale reuse and those that support large-scale reuse; here the interest is focused on the scale of reuse envisaged in two ways: scale of coverage throughout the software development process and scale of reusable products from the process. This allows reuse to be considered in the overall context of research concerned with improving the development process for software products and allows both composition and generation to be considered in combination.

The software life cycle gives an abstraction of the software development process; through a process of refinement, ideas are transformed into programs [95]. Lehman *et al* have characterised this process of refinement as a chain of linguistic transformations. The first step is the abstract formulation of conceptual requirements; and the final step is the concrete realisation of the software as executable code. Thus in describing software reuse, an axis of abstraction associated with the stages of the development process can be identified, and reuse of software products can take place at these various levels. Along side this, there is a scale of granularity. So with respect to design reuse, at the large end of the scale is the reuse of complete system designs while at the small end is reuse of simpler constituent component designs. IPSEs that capture and preserve representations of the software at each stage in the life cycle provide the basis for reuse of software at the appropriate level of abstraction and granularity. Hence the link between IPSE development and reuse as found in the Alvey programme and in the ECLIPSE project discussed below.

Developing a systematic approach to programming led many to look for ways in which the activity of programming could be automated in the seventies. An important aspect of this work came from applying approaches from Artificial Intelligence to Software Engineering. Two major reuse research projects of long standing can be seen as developments emerging from these concerns: Draco (University of California, Irvine) and The Programmer's Apprentice (MIT). The Draco project placed its emphasis on the development of domain languages obtained through a process of domain analysis. From the domain languages, general design models are abstracted and these form the basis of specific application models obtained through a process of transformation. The Programmer's Apprentice project placed its emphasis on a

library of plans obtained through a process of transformations performed on existing programs. From the plan library, a plan is selected and then transformed into a program. The plans are all represented using a uniform representation, the Plan Calculus, a combination of data and control flow, and the predicate calculus.

In both projects, transformational approaches were used, i.e. software descriptions at a high level of abstraction, general design models and plans, were refined via a chain of linguistic transformations into implementations. Uniformity of representation was achieved through translation. Both these projects resulted in the development of systems and approaches to reuse discussed in greater detail below. Both projects resulted in very limited transfer to industrial use; the effort required to develop domain languages and plan libraries with associated transformation systems is considerable. More modest approaches following Draco and the Programmer's Apprentice are also discussed below.

The Draco System

The Draco system and its approach to the construction of software from reusable components dates from earlier work on the transformational approach to software development [103]. The Draco approach is founded on the reuse of analysis information and design information in specific domains. This approach proceeds by a process of domain analysis to identify domain specific objects and operations followed by domain design of implementations of the objects and operations in *terms of domains already known* to Draco. It follows that underlying the Draco approach is a more fundamental analysis of the division of the universe of discourse into primitive and derived domains.

Reuse is effected by an analyst supplying a specification in a domain language already known to the Draco system as a result of previous domain analysis; and if there is a match between objects and operations required and those of a known domain, it is possible for a designer interacting with Draco to obtain implementations of these based on refinements of known designs. The designer decides which refinements to use and what kind of structure will result from the refinement chosen.

Refinements are recorded and may be replayed.

The Draco project was the first to recognise the importance of domain analysis to support reuse. Neighbors notes that building about 12 full usable Draco domains reinforced the idea that both domain analysis and design are very hard. The domains built spanned augmented transition networks, dictionary construction, generation of sentences in natural language, limited facilities for simulating parallel processing and a reuse module definition language; these are given in [65]. Freeman has provided a conceptual analysis of the Draco approach which relates Draco to particular software engineering principles such as abstraction, structuring, working systematically, modeling, compartmentalisation of knowledge and reusability [65]. The comprehensive coverage of Draco from the specification in a domain language down to the generation of code is arguably the reason why this approach has not had a greater success. The effort needed to develop high level domains in terms of known lower-level domains is considerable, and the lower-level domains need to be in place for the whole enterprise to work.

The Pegasus Tool also developed at the University of Calfornia, Irvine addresses the acquisition and reuse of software designs, and has been implemented in InterLisp [87]. It essentially consists of a database of software designs, an acquisition subsystem and a retrieval subsystem. The tool is able to handle descriptions in both text and graphic modes. The common description language used to represent designs is based on a subset of the Klone language, a knowledge representation language developed at BBN [37]. Retrieval is effected by means of a query language which allows the user interactively to formulate queries using database instances as templates. It is acknowledged that this system only addresses the design phase of the software development process, and the scale of designs considered is small, for example, sorting algorithms. This approach while much more modest than that of Draco does not appear to have been subjected to any reported application. The tool developed simply provides the mechanisms for supporting design reuse; the difficult task of actually developing the descriptions of the characteristics and behaviours of software concepts and artifacts is not addressed.

More recently, Freeman and Prieto-Diaz have addressed the classification of software for reuse [123]. Prieto-Diaz and Freeman have constructed a prototype library system based on the classification system reported in the above cited paper with about 200 program descriptors; this system is a UNIX based implementation using the University of California's Troll/USE prototyping tool as the underlying database system. The system does not directly support the phases of adaptation and composition although the system attempts to rank reusable components using a reuse effort estimation metric. This prototype has been the basis of a scaling-up exercise to produce a version for use in a production environment at GTE Laboratories [127] to support classification and selection of code components.

The Programmer's Apprentice

It has long been recognised that the techniques of Artificial Intelligence particularly those of Intelligent Knowledge Based Systems offer some scope for codifying the expertise of programmers and automating some aspects of the software development process. Charles Rich and his colleagues at MIT for several years have been engaged in such a programme of research known as The Programmer's Apprentice [135, 136, 134, 137]. A crucial part of this research has been the development of an appropriate formalism for representing software components; to this end, the project developed the Plan Calculus. The Plan Calculus allows extended flowchart schemata to represent algorithms and data structures as well as sets of axioms in the predicate calculus. In the Programmer's Apprentice implementation, plans are stored in a plan library indexed by the specifications, and their relationships with other plans. Two special purpose modules have been implemented to transform plans into programs and programs into plans. This research has been rather narrowly focused on the reuse of plans within the Lisp programming community although in recent years the focus has been broadened with work on The Requirements Apprentice [133]. It remains for this approach to be realistically applied in some particular application domain. The effort required to do this does not appear to have been considered by the authors. Most recently this group has introduced research towards developing The Designer's Apprentice [166]. Here again, the effort required to scale up this approach towards a practical application is not considered.

ECLIPSE, DRAGON and CAMP

Three other projects developing models, methods and tools to support reuse from this period are the ECLIPSE project, the DRAGON project and the CAMP project. The approaches taken by these projects are discussed below.

As part of the UK Alvey Software Engineering research programme, the work on the ECLIPSE project was in part explicitly concerned with developing methods and tools to support the reuse of existing software. Sommerville and Wood's work on classification has been the basis for a prototype implementation of a components catalogue [149]. This prototype has been developed using Prolog and catalogues a set of software components from the UNIX operating system. The system components are a dictionary of 300 verbs and nouns describing UNIX component functions and objects, a library of 300 or so component descriptions manually produced from the UNIX documentation, a user interface which allows the user to form requests by inputting keywords into skeleton frames, and the pattern matching strategies implemented in Prolog. This system is concerned with effective classification to support improved retrieval. Representation and composition issues were partially addressed by the development of SySL (System Structure Language) (already mentioned and reviewed in more detail in chapter 3). Interfacing of reusable software components was addressed by the development of the Component Description Language (CDL). This was largely an experimental vehicle offering a mechanism for designing large Ada systems at the component level [36].

Some aspects of developments in the ESPRIT Dragon Project can be traced to reuse research initiated in the ECLIPSE project such as the library and classification scheme described above. Tools were developed to support access to a software component catalogue and component information store. These are coupled with a design support tool, the Designer's Notepad [148]. This tool supports the description of informal linkages amongst the retrieved components and any other elements of the design. The Dragon project embodied a transformational approach to reuse [94]. All the tools developed by this project are general purpose. The tools were customised to support components represented in DRAGOON, Distributed-Reusable-Ada-OO-

Notation, as part of the research within Dragon.

The CAMP (Common Ada Missile Packages) Project [12] consisted of three distinct phases:

1. CAMP-1 Feasibility Study (Sept 84 - Sept 85): Identify commonality in missile flight software systems, design parts, and design supporting Parts Engineering System (PES). Ten missile flight software systems were studied; 454 parts were identified. A parts taxonomy was also identified; two classes were distinguished: domain dependent parts and domain independent parts. In CAMP, a part is an Ada package, subprogram or task usable in a stand-alone fashion.

2. CAMP-2 Implementation and Demonstration (Sept 85 - May 1988): The CAMP software developed was distributed to over 125 USA government agencies. Some measures of increased productivity were obtained when the CAMP team re-engineered an eleventh missile flight software using the PES and parts. The PES provides designers with three main functions:

   - identification function: either an application approach or an architectural approach may be used,
   - cataloguing function: this covers catalogue maintenance and catalogue locating,
   - constructor function: this is based on design templates for parts and rules for part construction.

   The application approach to identification involves the engineer giving specific features of the missile; whilst the architectural approach allows the engineer to step through a missile model represented by the hierarchical missile parts taxonomy. Twelve constructors were identified and made available: Kalman filter, data types, finite state machine, navigation subsystem, navigation component, pitch autopilot, lateral directional autopilot, data bus interface, task shell, time-driven sequencer, event-driven sequencer, and process controller. The aim has been *to generate customized (i.e. application specific) software components from standard designs.*

3. CAMP-3 Refinement and Technology Transition (July 1988- October 1990): As part of this work, thought was given to means of ensuring that the reuse is practised. Three steps were recognised:

   - identify more domain-specific parts,
   - establish reuse guidelines,
   - require reuse if possible.

   It was recognised that the order is important. Requiring mandatory reuse before guidelines and parts are available is useless.

The fact that this project chose a very limited application area, i.e. missile software flight systems, and succeeded, has influenced the work reported in this thesis to restrict itself to a specific domain in applying the approach to describing software concepts that has been developed. CAMP considered both classification and composition through provision of a hierarchical missile taxonomy developed to give guidance in the part identification and in the part construction from the constructors. The productivity tests made by CAMP are relevant to domain analysis. They found that as time progressed with experience that they got substantially better at producing new parts for their PES. The CAMP project can be seen as applying many of the initial concepts of how to carry out domain analysis as formulated in the Draco approach; however, the focus of this work remained small scale only considering the reuse of parts rather than whole system designs. CAMP gave consideration to the system context for reusable parts through its hierarchical taxonomy, but this was seen as a framework for guiding development and classification of parts rather than as an object of reuse in its own right.

Summary of Approaches to Reuse Support

The paradigm of component reuse is overwhelmingly the dominant view of reuse support approaches. All of the above provide examples of component based approaches to software reuse. In these approaches, the software engineer is assumed to be the principal agent developing the system architecture. This may take place

before the designer searches for reusable components, or alternatively the designer may develop the architecture based on knowledge obtained about existing components. This thesis will argue that more than component reuse based approaches are needed to support the designer in the very early stages of design. The research carried out in support of this thesis is focused on moving away from the component based approaches to reuse towards a more comprehensive form of reuse encompassing the reuse of the design frameworks themselves. In the work described here the emphasis has been on supporting reuse at the initial stage of design where the designer is searching for appropriate abstractions and structures to apply in the first formulation of design. This is an area largely ignored by the approaches reviewed above, but one which has been addressed within the Practitioner project discussed in the section that follows.

2.2 Specific Background for the Research within the Practitioner Project

The Practitioner Project was collaboration between Asea Brown Boveri AG and PCS Computer Systeme GmbH in Germany, Computer Resources International in Denmark, and three universities: Brunel University, the Technical University Clausthal and the University of Liverpool. The project completed its work in November 1991. The ultimate goal of this five year project was the development of a support system for the pragmatic reuse of software concepts; and a major result of the project was the development and demonstration of the Practitioner REuse Support System (PRESS). The project was concerned with the reuse of software concepts from designs through to code, focusing on concepts realised in existing software rather than the formulation of practices to be applied in the development of new reusable software.

Initially there were three major areas of concern within Practitioner:

1. description of software concepts, selection of appropriate forms of description to support reuse, and assistance with description by analysis of existing software documentation and source code;

2. classification, storage and retrieval of software concept descriptions to enable a software engineer to select the software design or code appropriate to meet specific requirements;

3. development of methods and tools to support the software engineer in the construction of software systems from reusable software concepts.

The research supporting this thesis contributed primarily to the first and third of these concerns.

The notion of a software concept as a potentially reusable object comes from accounts of software engineers' design experience; often, an engineer is able to formulate a solution to a new system requirement by following up informal ideas based on an understanding of existing software. These reusable ideas are the software concepts of the individual practitioner, i.e. software engineer; they may be fundamental ideas from Computer Science such as a *queue* or a *stack*, or operational ideas such as *order processing*, or very specific application ideas such as *material tracking in a strip processing line.* The working definition of software concept developed by the project [119] and used by author throughout this research is given below:

> *an abstract task, described by its purpose (and/or goal), the related objects and/or functional principles of the underlying mechanism (which will be typically, but not necessarily, of an algorithmic nature).*

At the highest level of description, a task is equated with an application. A software concept may be much less complex than an application; for example, the software concept, calculation of rolling forces, is an abstract task realised in software in the Steel Production domain.

Terminology analysis was the method used to establish the terms used in a particular domain to describe its software concepts. A faceted thesaurus was developed and subsequently used both in indexing of software concepts and in retrieval of stored concepts from a library.

In parallel with the terminology analysis, existing domain software was described using a structured form developed by the project, the questionnaire; development of the questionnaire is one of the main concerns of the research supporting this thesis. Although software concept descriptions are given a standardised structure via the project's form, use of a particular language or descriptive method to describe the concept is not dictated. It is assumed that the application oriented descriptions of concepts in the forms would be in the domain language, typically a natural language.

Over the first two years of the project during Phase I of the research, the emphasis was on collection of software concepts in specific application domains and investigation into appropriate descriptive methods to support reuse. Various approaches to reuse support were investigated, and two pre-prototype reuse support systems were developed using UNIX and the PCTE.

Phase II was concerned with the development of a prototype Practitioner REuse Support System (PRESS). In developing a reuse support system, the project was determined to reuse existing software concepts. A standard language for interactive text searching, CCL, has been used in the PRESS to support the searching for software concepts. The ISO Guidelines for thesaurus construction formed the starting point for the PRESS on-line thesaurus. The PRESS has been implemented using an existing database management system for the software concept store and thesaurus. The user views these through browsers built from a standard windowing system.

This prototype was experimentally evaluated during Phase III. The process of offer preparation was identified as one where reuse of software concepts could have an immediate impact. Responding to calls for tender, by preparing offers, i.e. proposals to build the required software, is an area where reuse typically has not been considered; and yet it provides a short enough illustration of reuse without being

too simplistic for demonstration purposes while at the same time bearing enough resemblance to the design process as a whole to make realistic use of the Practitioner methods and tools. In conjunction with experimental use of the PRESS to support preparation of offers, aspects of modelling the economics of reuse were studied.

In the final Phase IV of the project, the prototype PRESS was refined in light of the experience gained and in light of various research studies. Addressing the methodological aspects of populating the PRESS and employing it in the design of new systems - design-for-reuse and design-with-reuse - were the main concerns of these research studies. This thesis draws on the author's contribution to work within the Practitioner project. In particular, it is based on the research carried out by the author in the following areas:

- refinement of the questionnaire to describe software concepts,
- development of methods to support design-with-reuse and design-for-reuse,
- studies of designs in the domain of Steel Production and
- development of PRESS demonstrations.

A fuller account of the Practitioner project can be found in Appendix A. In particular, this describes in more detail the approaches to designing-for-reuse and designing-with-reuse which have been formulated and applied by the author in the course of this research. A more general account of research directions in reuse and the outstanding issues to be addressed by this thesis is given in the following section.

2.3 Identification of Research Directions in Reuse and Outstanding Issues

Four main areas of research in reuse can be identified and characterised in terms of design as follows:

- Recognition/Abstraction (Reuse Representation) - Design-for-reuse-in-the-small
- Classification of Reusable Objects - Design-for-reuse-in-the-large
- Specialisation/Adaptation of Reusable Objects - Design-with-reuse-in-the-small
- System Composition with Reusable Objects/Reuse Process - Design-with-reuse-in-the-large

With respect to the models, methods, tools being developed in the area of reuse research, there has been a move from *ad hoc* reuse towards more systematic reuse; that is, reports in the literature have shifted from accounts of how reuse has been accomplished towards formulation of rules and guidelines and more speculatively towards principles and theories of reuse, e.g. see [79, 16]. Many early research reports were on the experience of reuse; and now there is more concentration on understanding and improving the reuse process. Several reuse models, methods and associated tools have been developed as reviewed earlier in this chapter. However, these approaches to reuse with the exception of the Practitioner project have concentrated on component based reuse and ignored the reuse possibilities at the initial stages of design formulation. Ultimately, this thesis will show that examining reuse potential here as well as throughout the development of software will lead to a better overall understanding of design.

A key research issue is the role that representations play in facilitating reuse. It is recognised that standardised languages allow communication of collective experience and knowledge in the Software Development Process. Specialised languages whilst bringing power of expression may impose barriers to widespread software reuse. Translation may not be feasible; discontinuities may exist. There have been many language specific approaches to reuse, for example, many reuse projects have developed reusable software in Ada; of the projects reviewed here, the ECLIPSE reuse work, Dragon and CAMP had this concern. A fuller review of Ada reusability efforts can be found in [156]. The promise of formal methods as a basis for universal representations supporting reuse remains to be fulfilled although this continues to be an area of active research, for example, Tracz' LIL based development, LILEANNA,

combining Gogeun's proposed approach to reuse with the specification language developed for Ada, Anna [155]. Work on very high level representations for design reuse is reviewed in more detail in Chapter 3; this research holds out the promise of supporting reuse early on in the development lifecycle especially during the phase of design known as conceptual design [110] when the designer must attempt to determine the initial structure of the system at various levels of abstraction.

Many outstanding issues in reuse research remain to be tackled. The work described in this thesis is concentrated on two areas:

- Recognition and Abstraction, and
- the System Composition and the Reuse Process.

Better means for describing generic software concepts, both structure and function, are required. Systematic approaches to abstracting concepts from existing software need to be developed; the development of domain analysis techniques to facilitate reuse provides a starting point. A helpful characterisation of domain analysis has been provided by Biggerstaff:

> *Domain analysis is the building up of a conceptual framework, informal ideas and relations: the formalization of common concepts.*

(quoted in [157]).

Much more work remains to be done formulating a systemic approach to domain analysis, but the foundations are becoming established [128].

The description of software concepts requires conversion of application domain specific knowledge into agreed descriptive theory. Often such theory is unarticulated in a particular application domain; or worse, no such codified knowledge exists. Here work on specification discussed below is relevant.

Better means are required for describing the compositions of components to form software systems and for describing the derivation of specific instances from generic models.

A better understanding of software construction is required. The distinction between horizontal and vertical composition in the development process and the possibility of reuse at a variety of levels as described by Goguen [69] requires demonstration. The notion of software evolution is a promising basis for theory which will allow more control to be exerted over the software development process.

It can be seen that better means of description are required to support reuse; intuitively, this follows from the primacy of language in the history of ideas. It can be related to more general concerns in software engineering to do with specification of software systems as discussed below.

Several authors have compared the activities of program description and theory construction. Zemanek makes this point quite clearly by recalling the development of Wittgenstein's philosophical studies of language [174]. While it is the case that the computer system perfectly realises the world of the **Tractatus**, it is only in the context of a particular application domain that a program has a meaning, as Wittgenstein realised later in the **Philosphical Investigations** with his detailed study of how words are employed in various language games. Wittgenstein's insight here is a development of Frege's motto:

> Only in the context of proposition does a word refer. (*Nur im Zusammenhange eines Satzes bedeuten die Worter etwas.*)

[66]

Zemanek summarises this view as follows:

> *There will be forever a gap between the formal universe in our systems and the informal reality, between the domain of programs and algorithms*

and the life of people and communities - a gap which remains to be bridged by the human being, before and outside the mechanical and formal tools.

In their paper, "Putting Theories Together to Make Specifications", Burstall and Goguen [42] arrive at the following speculative conclusion that the main intellectual task of programming is elaborating the theories which describe all the concepts used in the actual program. This paper is an informal introduction to the development of the Clear language intended for describing theories in a structured manner. Burstall and Goguen speculate that while Clear is intended as a tool for program specification, it could also be used to represent knowledge.

Maibaum and Turski reiterate and develop this point that specification-building is similar to theory construction [158]. In so far as the specification represents in abstract terms a view of the application domain, the specification binds the program to its application domain. They present a programme for converting domain specific knowledge into a descriptive theory and program specification.

Below is a list adapted from one given in Maibaum and Turski which outlines the main steps involved the conversion process:

1. Catalogue all concepts in terms of which the domain-specific knowledge is expressed.
2. Categorise the catalogued concepts with respect to their functionality (find out which ones are constants, relations and functions).
3. Establish structural and hierarchical dependencies between the categorised concepts.
4. Discover which concepts are primitive and which are derived.
5. Check if all primitive concepts are indeed included in the catalogue.
6. Check the consistency of the resultant descriptive theory and if possible its completeness.

This list in brief provides a basis for the work of studying how to describe software concepts in a particular application domain. Although these principles have been developed with respect to the specification of new software, they are equally applicable to studies of existing software in order to understand and determine the concepts on which it has been based. The work reported here seeks to draw on these guidelines and apply them in the description of software concepts embodied in existing software systems. The process outlined by Maibaum and Turski is preparatory to making a formal specification of a software system. Formally re-specifying the software concepts has not been part of this research. The studies reported here have been based on describing software concepts as realised in existing software already specified, but often lacking a clear identification of its underlying concepts.

2.4 Conclusions

This chapter has reviewed general research in reuse and specific research projects concerned with supporting reuse with particular emphasis on the Practitioner project. Its aim has been to provide the background and motivation for the research of this thesis. Four areas of reuse research have been identified, and the concern of this thesis have been located within these as being primarily one of reuse representation to support system composition at the conceptual level of design.

In the following chapter, consideration will be given to the requirements that supporting software concept reuse places on a language and a development of the questionnaire to meet these requirements will be presented. Subsequent chapters will be concerned with the application of this work and an evaluation of its usage in practice.

Chapter 3

Representation of Software Concepts for Reusability

This chapter discusses in greater detail the form of software concept descriptions developed to support software concept reuse. The requirements for a language to support software concept reuse are elaborated and the form of software concept descriptions developed as part of this research is discussed in the light of these requirements in order to establish its adequacy.

3.1 Introduction and Overview

To support reuse of software concepts, a representational form with a certain amount of minimum expressive power is required. Here an elaboration of what is required will be made and it will be shown how the concept description form, the CDF, can be considered as such a representational form. The case will be made for classifying the CDF as an Interconnection Language - a class term that will be used here to cover languages based on developments of the Module Interconnection Language (MIL) concept originally proposed in [54]. A short survey of interconnection languages is

included here as background. Two developments of MILs that have influenced the development of the CDF: Goguen's proposal for a Library Interconnection Language, LIL [69] and the System Structure Language, SySL [146, 149], are examined. Both of these developments arguably could provide the expressive power needed to represent reusable software concepts, but each has drawbacks which will be discussed below.

Other contemporary work on languages to support reuse is also reviewed, and this provides an additional basis for comparison to the approach taken in developing the CDF. This work includes the framework architecture language and the simple component description language, Alfa, investigated by Henderson and Warboys [75, 76]. Their work is relevant to the discussion of the CDF because of its support for the abstract description of system architecture. There is also a close resemblance between the CDF as a canonical form for the description of software concepts to support their reuse and the notion of the *canonic software component* defined in the Pi-language using the concept of a concurrently executable module (CEM) and Weber's approach to describing the integration of reusable software components [167].

3.2 Background: Review of Interconnection Languages

Interconnection languages provide the conceptual background to the development of the CDF; these are reviewed below, and two of these which particularly influenced the CDF's form are described separately in greater detail using the same example, i.e. the concept of order processing in a strip processing line.

The composition of systems from existing components through a process of interconnection was first explicitly addressed by DeRemer and Kron [54] in the mid-seventies with their formulation of the basic principles of a Module Interconnection Language (MIL) and their recognition that such a language allowed large systems to be pro-

grammed by means of specifying the interconnection of their parts independent of the programming of their parts, that is, modules. DeRemer and Kron characterised this distinction as Programming-in-the-large versus Programming-in-the-small. Programming in the large requires more than the separation of specification of modules interfaces from their corresponding module implementation as realised in several languages developed in the the late seventies, notably Wirth's Modula and the Ada language; in addition, it requires a means of separately specifying the system architecture, that is, the interconnections or structure of modules to form a composite system.

A related concern in the development of systems is their effective construction; this led to the development of software build systems such as Feldman's pioneering **make** [62] tool which enables regeneration of software systems from separately compiled modules. Tichy extended MIL principles and brought them together with developments in configuration management and version control techniques [154]; this work has been the basis for various support tools developed as part of IPSE projects: the Gandalf project at CMU [68], the Adele project at LGI, Grenoble [61] and the ECLIPSE project in the UK [146]. A more thorough survey of MILs with examples has been provided by Prieto-Diaz and Neighbors [124]. The major contribution of MILs is that they provide a means of expressing the architectural design of a system and can be used to control the configuration of specific system implementations.

A development of MIL principles by Goguen [69] has been to propose the use of MILs not only to assemble code components, but also to assemble modularised formal specifications of components. Goguen has made an important distinction between horizontal and vertical composition. MILs address horizontal composition of systems, that is connection of components at a given level; vertical composition is concerned with moving between levels of abstraction, say from the system design to its implementation. Goguen's insight has been to integrate transformational design, that is vertical composition, with horizontal structuring. Here by transformational design, we mean the view of programming as a series of *linguistic transformations* [95] from a high level specification to an executable form.

Within the ECLIPSE project, Sommerville and Thomson have developed the system structuring language, SySL [149]. Like LIL, SySL allows the description of systems at various levels of abstraction to describe the static structure of systems. These two languages, LIL and SySL, are discussed below in greater detail as they have influenced the development of the CDF for representing the structure of software concepts.

3.2.1 Goguen's Library Interconnection Language: LIL

Central to Goguen's proposal of LIL is the need for a common formalism to underlie the integration of the multitude of entities associated with the software development process: code, documentation, requirements, specifications, designs, project data, etc. The advantage of such a common ground is that it ensures representations and assumptions about them will be compatible. As Goguen notes, Cohen and Jackson argued strongly that the ESPRIT Programme should be based on a common formal ground [50].

Goguen has proposed a library interconnection language, LIL, as means of assembling large programs from existing entities; thus reuse is achieved by interconnecting instances of the same entity in more than one program. Reuse in the context of a development environment with a database of entities ranging from requirements, specifications, design histories, code to project status information is not simply limited to reuse of code components. In Goguen's proposal integration of these diverse entities is achieved by describing them all in a common formalism: LIL.

LIL provides for formulation of high level specifications and description of general parameterization mechanisms. An important feature of LIL is that through the use of theories, semantic descriptions consisting of either formal or informal axioms, may be associated with entities or entity interfaces.

There are three major semantic concepts which characterise LIL:

- theories which associate semantic descriptions (either formal or informal) with components
- views which describe semantically correct bindings at software interfaces
- composition, both horizontal and vertical.

In LIL, theories declare the properties that an actual parameter must have when substituted for a formal parameter. Views express that a given entity satisfies a given theory in a particular way. Explicit connection commands are used to express how to compose a system from its components. The basic LIL entity is the package; it is structured like a theory but unlike a theory actually has implementations associated with it. In LIL, packages and theories may both be parameterised.

Below is an example albeit incomplete illustration of LIL where it is used to describe an Order Processing System in a Strip Processing Line (SPL below).

```
theory ORDER is
types ORDER_TYPE
functions
...
vars
...
axioms
...
end ORDER
view SPL_ORDER_DEF :: ORDER ⇒ SPL_ORDER is
– defines a view of SPL_ORDER as an ORDER
types (ORDER_TYPE ⇒SPL_ORDER )
end SPL_ORDER_DEF
generic package ORDER_PROCESSING[ORDER_TYPE :: ORDER] is
– order processing package with interface requirement ORDER
types
functions
...
vars O : ORDER; ...
axioms
...
end ORDER_PROCESSING
make ORDER_PROCESSING_IN_A_SPL is
ORDER_PROCESSING[SPL_ORDER_DEF] end
– here we instantiate the parameterized ORDER_PROCESSING
– with the actual parameter SPL_ORDER_DEF
– which defines a view of SPL_ORDER as an ORDER.
```

In the above example, horizontal composition is achieved with each *theory* and *package* description by decomposing the description into the parts listed such as

types, *functions*, *vars* and *axioms*. Vertical composition is achieved through the use of *make* and *view*.

3.2.2 The SySL - ECLIPSE System Structure Language

The SySL language developed as part of the Alvey ECLIPSE Project allows the description of software structure in terms of its components and is supported within the Eclipse Integrated Project Support Environment by various tools. These tools and the SySL language have been described in [146, 149, 148]. The account given here is based on [149].

In SySL, systems are described by use of a class construct; and structure is described in terms of dependencies on other subsystems. Subsystems may themselves be described by further classes. A system description in SySL consists of two parts:

- a generic part defining the class and
- a descriptive part i.e. the instantiation of the generic part.

The first part may be viewed as a template for a number of potential systems; thus, SySL may be used to describe *software families*. The instantiation of the generic part gives details of the specific software components which comprise the system and their interfaces. Assertions are used to associate attribute information with classes or particular components; for example, these are used to express constraints such as both optional components in a class cannot be absent.

SySL allows the user to distinguish between **systems** and **components**; components need not have a structure. Interfaces to both systems and components are defined by means of a **provides** clause and **requires** clause with an additional **external** clause to indicate dependencies on components defined elsewhere required for configuration.

Below is an example of an incomplete SySL description of an Order Processing System in a Strip Processing Line. This example illustrates the structuring features of SySL discussed above. Note that conventionally in SySL, class names are given in capital letters.

```
class ORDER_PROCESSING_SYSTEM is (order_processing_system_in_strip_process_line)
structure ORDER_PROCESSING_SYSTEM is
DIALOGUE_SYSTEM,
COMMUNICATION_PROCESSOR,
ORDER_PROCESSING,
DATA_RECORDING,
GROUP_LEVEL_CONTROL,
MATERIAL_MANAGEMENT
end structure
structure ORDER_PROCESSING is
...
end structure
system order_processing_system_in_strip_process_line : ORDER_PROCESSING_SYSTEM is
DIALOGUE_SYSTEM ⇒ operator_interface,
COMMUNICATION_PROCESSOR ⇒ data_communication_subsystem,
ORDER_PROCESSING ⇒ order_management,
DATA_RECORDING ⇒ reporting_and_logging,
GROUP_LEVEL_CONTROL ⇒ set_point_processing,
MATERIAL_MANAGEMENT ⇒ material_tracking
end system
component order_management : ORDER_PROCESSING is
...
end component
```

SySL does not treat systems and components in an orthogonal fashion; SySL does not appear to address the forming of component classes even though this could be easily done with the machinery SySL already has for class definition, i.e. explicit enumeration.

It must be admitted that SySL is not intended for the description of system structures to support their reuse. Its intended area of application is as an aid to system understanding during the development and construction of large software systems.

3.2.3 Related Language Developments to Support Reuse at a High Level of Abstraction

With their development of the framework architecture language, Henderson and Warboys aim to address the needs of the system engineer who must take two distinct

views of the system to be developed: one in terms of the business needs to be satisfied and the other in terms of the technology with which they can be met [75]. This is a simplification as in many domains both the business needs and the technological solutions may be themselves expressed from several viewpoints [41]. Where the reported work of the Practitioner project [22] strongly supports Henderson and Warboys in their assertion that a common language is required with which to communicate proposed solutions to those who need to be convinced that they satisfy their business needs and to those who will be charged with the task of realising the solutions technically, i.e. the implementors. They stress that it is also necessary that the language be at a very high level of abstraction, i.e. is very noncommittal about design details. The basic notion at the heart of their language is that of a component as a provider of services. Primitive components are defined in terms of the services they require and supply. The language they propose expresses systems in terms of the services they provide using "+" to denote horizontal composition and "," with square brackets to denote vertical compositions.

Henderson and Warboys attempt to ensure that their language is sufficiently formal to allow a simple calculus of manipulations of system structure descriptions to be made. They propose that existing systems be analysed and described using the Framework Architecture approach and that these descriptions be exploited in determining the potential for component reuse. Through use of their flatten operation, they remove system structure and reusing system structures does not appear to be part of their approach even though they do provide a means of representing hierarchical structures and composing subsystems in hierarchies.

In another paper, Henderson and Warboys describe Alfa [76]. In Alfa, as above, components are defined in terms of services that they require and supply; and systems are defined as structures of components. Alfa appears to be a development of the earlier framework architecture language. An innovation is the introduction of generic components to describe any one of the possible versions (or implementations) of a component; thus the required interface of a generic component is the maximum interface that will support whichever actual version instantiates it and the supplied interface is the minimum. A mechanism for treating structured com-

ponents and generic components as "Black Boxes" is also introduced; this enables a determination whether or not one component can be used to implement another, i.e. by supplying at least the services supplied and requiring at most the services required. In Alfa, interfaces are described using C++ function prototypes which the authors found adequate but not ideal. It is noteworthy that they would like to return to a more abstract view of interfaces.

Another development to support reuse at a high level of abstraction can be found in Weber's notion of the *canonic software component* defined in the Pi-language using the concept of a Concurrently Executable Module (CEM) and Weber's approach to the integration of reusable software components [167].

Weber contends that integration is central to the concept of reuse and focuses on establishing the technical basis of integration in terms of canonic forms for describing reusable software components and architectures. Weber views integration as consisting of an analysis of parts and their subsequent synthesis into the new whole. In the paradigm formulated, software integration must be based on provisions built into software for its later integration. This is the basis of a properly engineered integration; insufficiently prepared software can only be integrated after its proper reengineering. Analysis of software parts presupposes that software is properly structured, specified and documented. Structuring provides the basis for decomposition into component part; specification is an aid to understanding functionality; and documentation provides for different realms of reasoning about the system from different angles. Weber claims that *not all - and sometimes even none - of these provisions are really built into existing systems.* For synthesis of software, an integration framework is required. This provides for both interconnection defining static (i.e. structural) relations and interoperation defining dynamic relations during execution. Weber introduces one type of these - a software "component model" and explains how it supports the reuse of software. Weber's work has been included here because his emphasis in the software component integration framework is on structural relations.

In Weber's concept of reuse, the integrated component, C with functionality S, is

the result of integrating components, *C1* with functionality *S1*, and, *C2* with functionality *S2*, under the integration framework, *i*, where the integrated component *C* serves as the integration framework. This relationship is expressed as follows:

$C(S) = \{ C1(S1), C2(S2) \}$ under i.

Weber introduces term canonic software component to denote an abstract entity that exhibits the properties common to all components ever to be reused. Weber considers that any development of understanding of functionality requires a formal denotation of the semantics for each component. Thus he bases his canonic software component on the concept of a Concurrently Executable Module (CEM) as defined in the Pi-language.

He argues that such uniformity in describing software components is essential in large-scale software production with reuse. Without it, analysis and synthesis would be unmanageable and not amenable to automation. He acknowledges that currently as no canonic form of semantics exists, reusers must rely on hard-to-understand domain semantics. He characterises practical software reuse in industry as a mixture of "constructive" means, i.e. *a priori* denotation of interfaces, and analytic means, *a posteriori* determination of existing component interfaces. Here his characterisation appears to follow the distinction that will be made below in discussing reorganisation of software concepts and interface checking between synthetic construction and analytic reconstruction.

3.3 Requirements for a Language to Support Reuse of Software Concepts

There are two important aspects of reuse which must be expressible in a language aiming to support the reuse of software concepts in design:

- reuse of software construction principles via generic models of system architecture, and
- reuse of software theories (either formal or informal) via generic components.

It is of course also important that any language covering the generic cases will also support the reuse of specific system architectural and component concepts. For example, when designing a compiler, a software engineer can reuse the established principles of compiler construction and these give the designer generic models or frameworks of compiler design into which various component design concepts such as the lexer, the parser, code generator, optimizer etc may be slotted. These component design concepts may themselves be pre-existing i.e. reusable components. In the case of a particular component, say the parser, its design may have been automatically derived as a result of well understood theory, in this case, that of parser generation. The software engineer might also design the compiler working within very specific constraints, for example, to reuse an existing back-end for code generation within a family of existing compiler models.

From the above, three requirements can be identified, as follows:

R1 a means of describing generic system models and relating specific system designs derived from a generic model,

R2 a means of describing generic component concepts and relating the derivation of specific components from generic components which have well understood theories (this derivation is an example of what Goguen calls vertical composition), and

R3 a means of describing the composition of reusable components to form the system design at a particular level of abstraction (Goguen's horizontal composition).

As the concern is to support the reuse of design concepts which can be described at various levels of abstraction as well as being structures of component parts at

any particular level, support in the language for describing vertical and horizontal composition is essential.

A more general way of expressing the requirement for vertical and horizontal composition is to consider that systems are built from parts. Once built, systems themselves can be conceived as part of some larger system. New systems may be designed in an evolutionary manner by small partial changes, i.e. new parts for old; or by developing new configurations of existing parts, or by a combination of these. Both Henderson and Warboys, and Weber in the papers cited above make the point that ultimately every system is a component in some higher level system. An essential element of any design language is support for the hierarchical whole-part structuring of system descriptions at various levels of abstraction.

In addition, as well as capturing the structure of composite concepts, a means of describing the interrelation of their parts at a particular level is required. For any particular concept, it is important to describe its external interface; and for a composite concept, as well as describing the internal interfaces of its parts, it is required to spell out the interface bindings i.e. to relate external and internal interfaces.

There are two distinct cases where the designer developing a new system design with the software concepts could benefit from some support for interface checking. One is where existing software concepts are brought together in a new composition; the other case is where one or more existing software concepts are substituted within an existing concept decomposition. The distinction between these two cases can become blurred, for example, depending on the number of substitutions. However, from the standpoint of interface checking, particularly where the designer's intent is to preserve the primary concept's overall specification (i.e. the latter case), it is an important distinction. In the first case, i.e. the case of a new composition from the component concepts and their given interfaces, it is possible to reason informally about the composition and the most appropriate interface bindings required to achieve the desired overall new system. When combining concepts, the designer must determine how the new concept's overall function can be achieved by interfacing the chosen set of concepts. In the second case, substitution of existing concepts in

a known decomposition, it is necessary to compare the existing interfaces' bindings needed with the given interfaces of the concepts to be substituted and to determine whether or not a reasonable interconnection is possible. In substituting each new concept within a given concept decomposition, the designer must check that the new concept has the appropriate interfaces to fit within the known decomposition. The first case can described as synthetic construction of a new concept and the second as analytic reconstruction of a new concept. These two cases are illustrated in Figure 3.1 - Two Cases of Concept Interfacing.

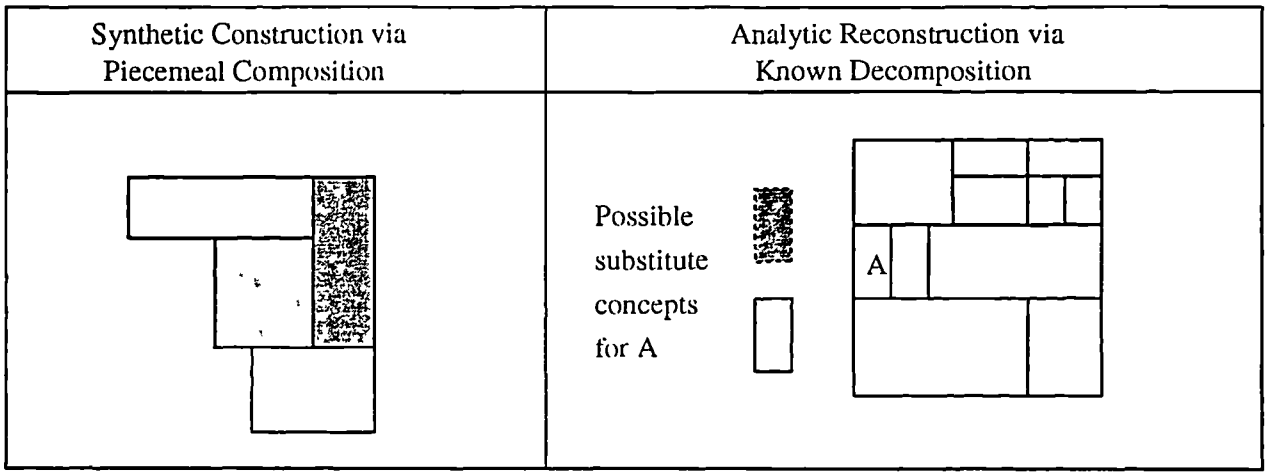


Figure 3.1: Two Cases of Concept Interfacing

It is also possible to describe the same concept from several different viewpoints where these viewpoints are related as different conceptual levels of abstraction. These notions, levels of conceptual abstraction, and horizontal and vertical structuring through composition, are not simply two ways of describing the same thing. Each design concept found at a particular conceptual level of abstraction may have its particular design from its specification down to its implementation, its own vertical composition, i.e. be described at various levels of abstraction in its design from its specification down to its implementation, as well as being horizontally composed of parts within these levels. To make this distinction clearer, in Figure 3.2 - 3-D Concept Levels of Abstraction with Vertical and Horizontal Decompositions, a figure from [152] has been adapted to illustrate how the concept of a modern computer

can be described as using three dimensions:

1. levels of abstraction in the concept description,
2. a hierarchy of whole-part descriptions, and
3. levels of abstraction going from the specification to the implementation; these constitute the levels of a particular design abstraction.

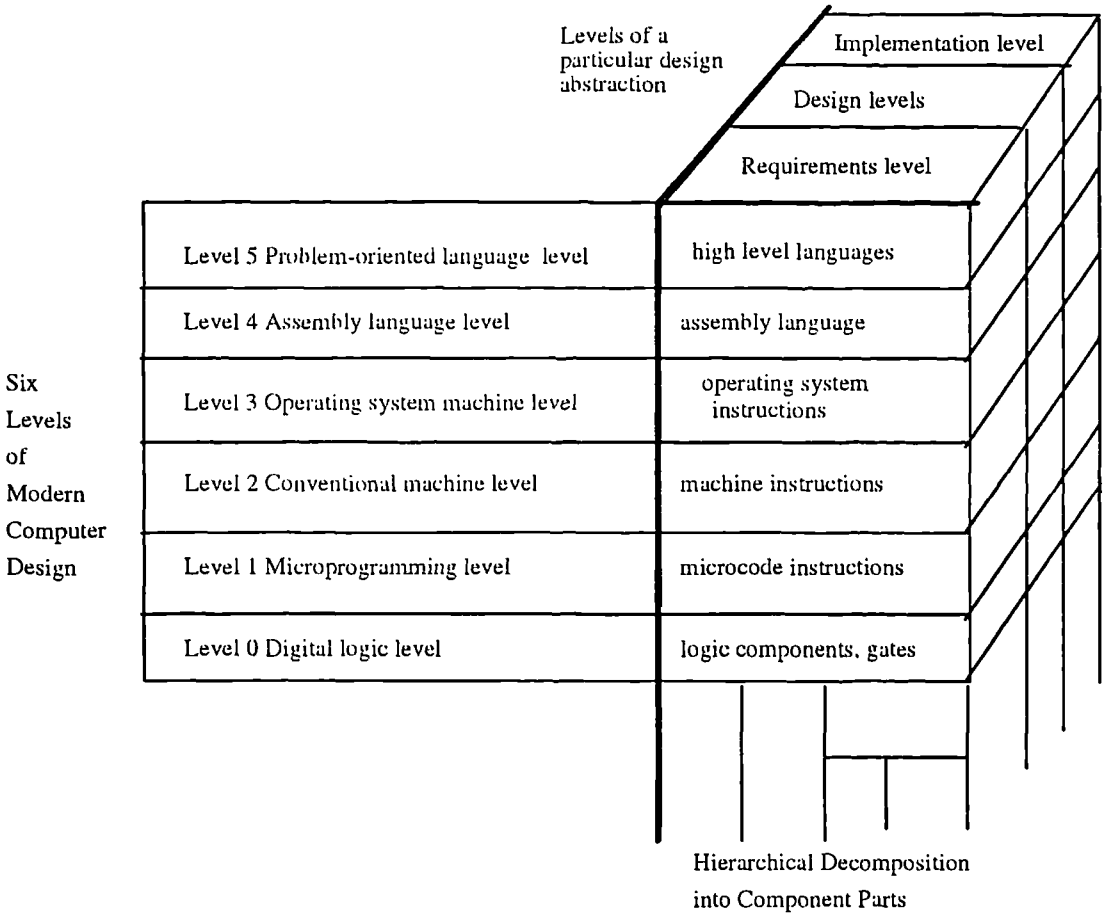


Figure 3.2: 3-D Concept Levels of Abstraction with Vertical and Horizontal Decompositions

First the concept of a modern computer is described from a number of viewpoints such as it is conceived by hardware and software designers at a number of levels - this gives the levels of abstraction to the concept. At each level, component parts can be identified and composed horizontally, e.g. basic logic components into integrated

circuits at the digital logic level; and in describing a particular integrated circuit, its functional specification can be given as well as its more detailed design and implementation forming its vertical composition.

The Practitioner project's goal of supporting reuse of software concepts as realised in existing software systems imposes a further requirement of providing a means of linking the elements of the software concept description with software life cycle documents associated with the existing software which embodies these concepts, so that the reusers have the possibility of improving their software concept understanding by refering to existing software documents including code.

From the above discussion, three further requirements have emerged giving a total of six requirements for the CDF, as follows:

R1 Generic Architecture Derivation a means of describing generic system models and relating specific system designs derived from a generic model,

R2 Generic Component Derivation a means of describing generic component concepts and relating the derivation of specific components from generic components which have well understood theories (this derivation is an example of what Goguen calls vertical composition),

R3 Concept Compositions a means of describing the composition of reusable components to form the system design at a particular level of abstraction (Goguen's horizontal composition),

R4 Interfacing Concepts a means of describing interfacing concepts used in the construction of a composite concepts at a sufficient level of abstraction,

R5 Conceptual Viewpoints a means of describing the design from different viewpoints where these may be related as conceptual levels of abstraction in the application domain, and

R6 Explicit Links to SLC products a means of linking the concept description with the software life cycle documentation if concepts in existing software are being described.

In the following section, it is shown how the CDF has been developed in order to meet the requirements identified above.

3.4 The CDF - A Standard Form for the Description of Software Concepts

Software concepts as realised in existing software systems are of course implicitly described by the software which implements them and its accompanying documentation such as requirements statements, specifications, detailed designs, manuals and other reference material about the software and its development, underlying concepts, etc. One of the aims behind the development of the CDF has been to conceive a filter through which the reuser could access relevant material needed to support the reuse of software concepts in the development of new systems.

The CDF is a development of a form for describing software concepts to support reuse, known as the questionnaire, which was proposed in [59]. The main entry headings of the questionnaire are listed in Appendix B. Initially as developed the questionnaire followed very closely the software documentation standard of one of the industrial partners' business divisions. This early form was found to be largely in line with best practice as described in the IEEE Recommended Practice for Software Design Descriptions [26]. The advantages of using a standard form of descriptions such as the IEEE Std 1016 or an internal company standard are that the information needed for planning, analysis and implementation as well as maintenance and evolution of the software system can organised and accessed easily. In addition, adherence to the standard form ensures that records of design details which have been preserved provide a common basis for reuse.

However, neither the early questionnaire form nor the IEEE Std 1016 explicitly recorded versions of concepts or the relationship between generic design concepts and specific instances derived from them. The original questionnaire only made

provision for describing a concept's historical development, mainly through links to reference documents. While this could provide a means of indicating the historical derivation, it was inadequate for relating generic concepts to specific concepts. There was also no way of relating alternative descriptions of the same software concept found in similar software systems. In CDF, concept versioning was introduced and the recording of the derivation relation which relates a particular version of a concept to its parent concepts was made much more prominent.

The questionnaire's entries for describing concept interfaces were based on identifying data inputs and outputs, control inputs and outputs, and reactions to exceptions. This level of detail was not recommended in the IEEE Std 1016 and in early experiences using the questionnaire it was found hard to supply without study of the detailed design. In the CDF, the specification of concept interfaces was streamlined as detailed below.

Both the questionnaire and the IEEE Std 1016 address the recording of the concept decomposition. In the IEEE Std 1016, the decomposition description records the division of the software system into design entities. In the questionnaire, there is an entry for recording a concept's immediate parts where these are those concepts (or functional entities) out of which the concept is composed. In addition, immediate parts are classified as either *active*, i.e. functions, procedures, tasks, etc., or *passive*, i.e. data items, files, etc. Here as with interfaces, the approach in the CDF development has been to streamline the description of a concept's decomposition and make it more explicit. The rationale for this again came from experience of applying the earlier questionnaire and the desire not to preclude decompositions based on object oriented approaches to design where the active/passive classification is difficult to make.

The deficiencies in the questionnaire identified above can be summarized as follows:

D1 no explicit recording of concept versions,

D2 no recording of relationship between generic design concepts and specific instances derived from them,

D3 historical development inadequate for relating generic concepts to specific concepts,

D4 no way of relating alternative descriptions of the same concept,

D5 interface description is too detailed,

D6 active/passive classification of immediate parts is restricting.

To overcome these deficiencies, during the CDF development, concept versioning was introduced and concept interface specification and concept decomposition description were streamlined. In addition, the recording of two important relations amongst concepts, concept derivation and concept decomposition, was made more prominent. These relations give rise to two important conceptual mappings for any particular concept version; a derivation map showing its history and a usage map showing its employment within other concepts.

Figure 3.3 gives an overview of the author's work relating to the development of the CDF. As a result of studying Interconnection Languages and other language developments to support reuse at a high level of abstraction, six major requirements were identified. In addition, as a result of analysing accounts of users of the Practitioner questionnaire and comparing it to IEEE Std 1061, a number of deficiencies in the questionnaire were recognised. The CDF has been developed from the questionnaire to satisfy these requirements and overcome these deficiencies. In what follows, more details of how the CDF supports the recording of these aspects of concept description will be given and the relevant parts and associated entries of the CDF will be related to the requirements identified earlier and the deficiencies listed above.

The CDF has four main parts; these are as follows:

CDF1. Concept Version and Derivation,

CDF2. Concept Specification,

CDF3. Concept Decomposition, and

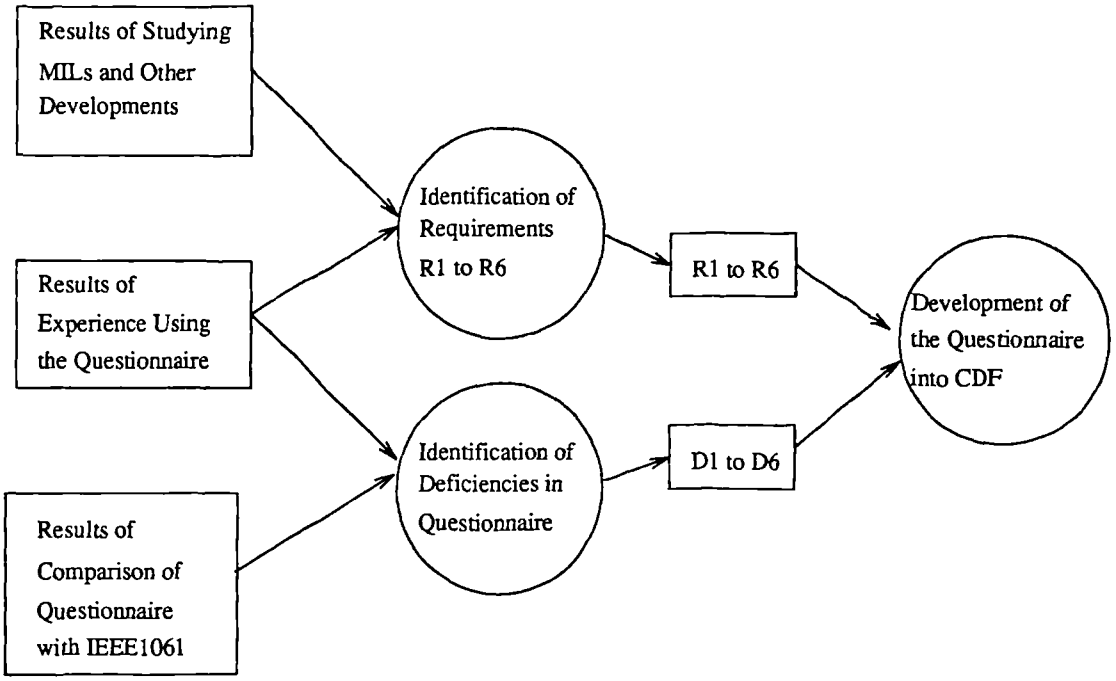


Figure 3.3: Overview of Work Relating to CDF Development

CDF4. Concept Links.

The latter two parts are optional. The Concept Decomposition is only included if the concept being described is non-atomic and has been decomposed in further concepts. The Concept Links are filled in if the concept can be linked to an existing implementation and software documentation and/or other references to giving a more detailed description. Note that in the list above and in the following discussion of the CDF, a link has been established with a concrete syntax developed by the author and given in Appendix C by prefixing each part of the CDF with the decimal numbering used in the concrete syntax of Appendix C.

The CDF supports the description of more than one version of a software concept through the use of version numbers. In the first part, the concept name and version number are recorded followed by a more detailed description of its derivation. Although the CDF records the concept name and version number, these may need to be revised by the administrator of the concept database when a particular CDF is

installed to ensure the integrity of the name space of the concept database.

From the reuser's point of view, a concept has a name and may exist in several versions each identified by a version number. Together, the concept name and version number identify a unique concept description, a completed CDF. Graphically, the CDF for the first version of a concept (V1 in the figure) can be unfolded as follows into four headings as shown in Figure 3.4. Note that this graphical form was proposed in order to allow the reuser to unfold the concept description recorded in the CDF to the level of detail required [31].

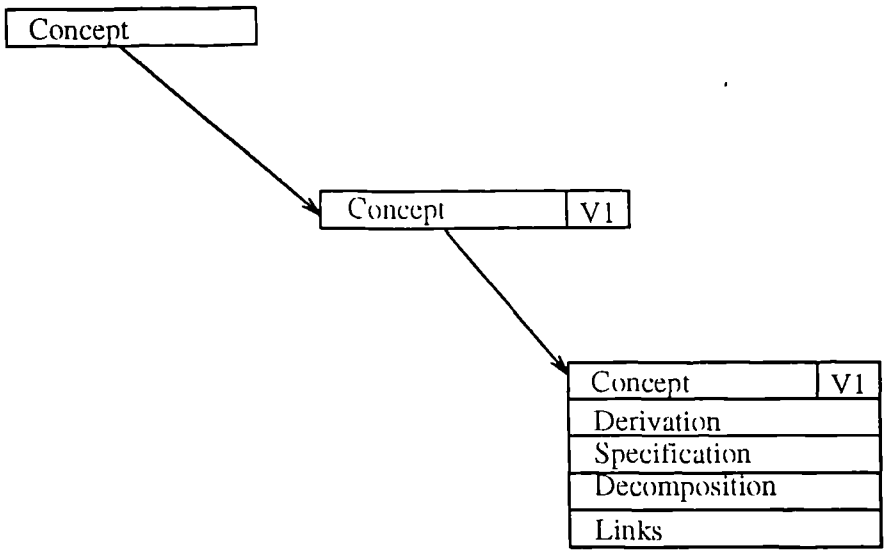


Figure 3.4: Progressive Revelation of Concept Details

3.4.1 Provision for the Recording of the Concept Derivation

The derivation covers the following aspects of the software concept:

CDF1.3.1 Description of Purpose,

CDF1.3.2 Authorising Person,

CDF1.3.3 Created by Person,

CDF1.3.4 Date of Creation or Entry,

CDF1.3.5 Source Concept Versions, and

CDF1.3.6 Creation Processes Used.

The first of these entries, Description of Purpose, is intended to record the requirements that gave rise to the concepts' development. It gives an account of what role of the concept plays in the context of a particular application domain. This information is helpful to the reuser in determining whether or not this concept may be useful in meeting the requirements for which the concept is being considered during design-with-reuse. The terms used in the entry may also be helpful in the retrieval process.

The entries concerned with recording the authorising person, the concept creator and the date of creation or entry in the concept base of the PRESS are not so relevant to assisting the reuser in retrieving or understanding the concept. They are used to give guidance on the accuracy and currency of the CDFs held in the concept base of the PRESS and allow the development of the concept base over time to be monitored.

The final two entries in this part of the CDF are related to the first two requirements noted above. The first of these allows the *derived from* relation to be established between a new concept and existing concepts. This relation is important for understanding the intellectual development of a concept. Typically these will be references to other concept descriptions. The second entry allows the intellectual processes leading to the development of a new concept such as generalisation or specialisation be noted. Tools used in creating the software concept such a parser generator or 4GL should be recorded. These may well be references to other concept descriptions. With these two entries, it is possible to establish relations between concepts and record more than their immediate description, as by using a set of CDFs related through either of these entries, a design history of related concepts can be described. It is possible to systematically trace the derivation of a concept through following the *derived from* relation amongst a set of CDFs by inspecting in turn the

concepts from which each concept has been derived. Such information is often overlooked in standard design description techniques, but is important to the potential reuser as an aid to understanding the concept. This is particularly the case where a generic version of the concept has been abstracted from several specific concept, and subsequently can be given as the source concept in their respective CDFs.

3.4.2 Provision for Recording the Concept Specification

The concept specification is concerned with describing what the software concept does and its interfaces that are required and provided externally. The following entries are found in this part of the CDF:

CDF2.1 Definition,

CDF2.2 Interfaces Provided,

CDF2.3 Interfaces Required.

The definition entry of this part of the CDF makes provision for recording the specification of the software concept either formally or informally. Where the concept specification has been parameterised this can be recorded under the entry, generic parameters. Provision is made for recording the following items under this entry:

CDF2.1.1 Function,

CDF2.1.2 Formalism,

CDF2.1.3 Generic Parameters, and

CDF2.1.4 Description.

Of these, only the last is required to be present in a filled in CDF as this is the most minimal definition of the software concept.

Through the CDF, external interfaces are recorded and distinguished as being either required or provided. The interfaces are simply listed under the last two entries by giving the name and version number of the relevant interfacing concept. These may be described in additional CDFs depending on the degree of domain analysis undertaken. Where additional CDFs have been completed, the subsequent interfacing concept descriptions can be either formal or informal. These external interfaces are bound to interfaces of the component concepts through entries in the Concept Decomposition described below.

A case can be made for giving formal semantics to the concept interface descriptions as this would allow interface checking as discussed below, but the case is not clear-cut. In the above discussion on requirements, two cases can be distinguished where interface description is required. In both cases where the matching of interface descriptions is required, using given CDFs, the designer must rely on the informal semantics of the interface names and the specific semantics of their associated concept descriptions if these exist; the thesaurus can help if the names and descriptions use related terms. Conventional typing of the interfaces is unlikely to be of much relevance, unless it is in terms of high level domain abstractions that have been standardised. For example, matching the interface types in terms of basic data types such as **real** and **integer** or UNIX tool byte streams is likely to be at too low a level. At this stage the interface semantics must be specified in terms that enable the designer to understand it at an appropriate level of abstraction compatible with the application concept's overall description. These considerations anticipate work in progress by Henderson and Warboys who acknowledge there is not yet a clear solution to the conflict between giving more formal semantics to interfaces and remaining sufficiently abstract to support high level design.

For example, consider the case of designing a compiler based on a given abstract machine defined by its abstract machine code, a form of intermediate code. Compilers using an intermediate code are commonly constructed in two parts, a front-end which translates the input source language to the intermediate code and a back-end which translates the intermediate code to the output target language. Informally, it is easy to determine that these two concepts, the compiler front-end and the compiler

back-end, can be interfaced. Detailed inspection may show that the intermediate code of the front-end does not quite correspond to that of the back-end. In this case, some adaptation of one or other of the concepts may be required: or alternatively, the designer may choose to introduce a bridging concept to perform the requisite code transformations. At the conceptualisation stage, such details need not concern the designer and the informal determination of compatibility of interfaces is sufficient.

Pahl and Beitz comment on the appropriate level of detail at which to consider interfaces [110]. Too low a level may exclude concepts that could be useful if the interface is widened. Thus they counsel against narrowing the interfaces early on in the design process. As the concern in the CDF is with supporting concept description, this is another argument for keeping the interfaces descriptions informal. More generally, the case for deferring the detailed description of interfaces can be made along the same lines as the general case for deferring decisions in design as long as possible [153].

However, the case is not so clear-cut. From a practical standpoint, empirical studies reported in [14] have shown that a high percentage (around 68%) of errors that appear at the system integration and test phase are program-module-interface errors [113]. Clearly, it would greatly assist designers if such errors were found much earlier on in the design process. It has been proposed that interconnection languages could solve this problem by taking their interface semantics from an underlying specification language; for example, see Goguen's LIL proposal in which OBJ is recommended for this purpose. This solution is open to users of the CDF provided descriptions in an appropriate specification language are available when completing the entries in this part.

3.4.3 Provision for the Recording of the Concept Decomposition

The CDF like the earlier questionnaire provides a recursive method of software description; concepts are decomposed into other concepts which are described by further CDFs until the concepts identified are atomic and can be defined directly without any further decomposition. In this way, a set of CDFs can be used to describe a concept that has a hierarchical whole-part structure.

In the concept decomposition part of the CDF, a separate entry is made for each component concept. The entry contains the following details:

CDF3.1.1 Concept Being Instantiated given by concept name and version number,

CDF3.1.1.3 Instantiation Paramenters or Specialisation,

CDF3.1.1.4 Purpose Served or Reason for Incorporation, and

CDF3.2 Interface Bindings.

The middle two items have been included to give the reuser insight into how a particular concept has been instantiated within another software concept and to indicate its purpose or reason within the decomposition. This information may be useful to the potential reuser in understanding why a particular concept decomposition has been made.

The interface bindings are used to links the interfaces of each component concept with the main concept's provided and required interfaces and to establish any internals interface links with other component concepts. This entry consists of two lists, as follows:

- external concept interface bindings, and
- internal concept interface bindings.

It should now be clear how the CDF supports both horizontal and vertical composition. Each concept is horizontally decomposed into its immediate component parts and their interfaces are described. Thus, the CDF can be used a means of describing the composition of reusable components to form the system design concept as required above, and as a means of describing interfacing concepts used in the construction of a composite concepts as required above.

In addition, each individual concept description given by means of the CDF can be vertically decomposed into a high level description of its purpose, a specification, a more detailed design and finally links to an implementation as detailed below.

3.4.4 Provision for the Recording of the Concept Links

This part of the CDF does not contain actual descriptions; rather it addresses the final requirement identified above as it provides a means of linking the concept description with the software life cycle documentation as well as other references which have been used in the descriptions found in the other parts of the CDF filled in for a particular software concept.

In this part, provision is made for recording the following entries:

CDF4.1 Code Module,

CDF4.2 Data Definition,

CDF4.3 Documentation and

CDF4.4 Test Package.

All of these entries are optional although where the software concept being described has been realised in existing software, establishing these links as a matter of record will meet the reuser needs which resulted in the above requirement.

Where the software concept being described has an associated implementation, the first two entries in this part of the CDF allow links to be established with the source code and any relevant data definition documents. The fourth entry allows links to any test package to be used with the concept. The documentation entry is used for any of the other software documentation or reference works which have been consulted in preparing a particular CDF. These links allow the reuser to follow up and reuse code if available as well as simply trace a particular CDF back to its original sources.

3.4.5 CDF Overview

One way of viewing the CDF is simply as a structured form for recording abstracts of existing software documentation concerning a particular software concept. In a sense, this is quite a valid view as in order to complete the various entries on the CDF for a software concept realised in an existing application, the domain analyst could abstract the relevant information from the associated software documentation. For example, in the CDF, the concept's description of purpose could be abstracted from the original requirements documentation. The specification given in the CDF could be abstracted from the original software specification. The decomposition into component concepts could be abstracted from the original high level design documentation. The links to existing documents in the final part of the CDF are not abstracts, but these enable any links to existing documentation to be made explicit. For example, the test procedures developed during the test activities of the construction phase could be used to complete the entry recording links to a test package. Figure 3.5 gives an overview of how work products associated with each phase of the software life cycle can be used in completing various entries in the CDF. This figure presents a simplified view of the software life cycle adapted from [98].

The CDF has an abstract syntax defining its structure independent of any concrete representation such as the graphical form developed by the author and used in Figures 3.6, 3.7 and 3.8. The CDF abstract syntax can be found in Appendix

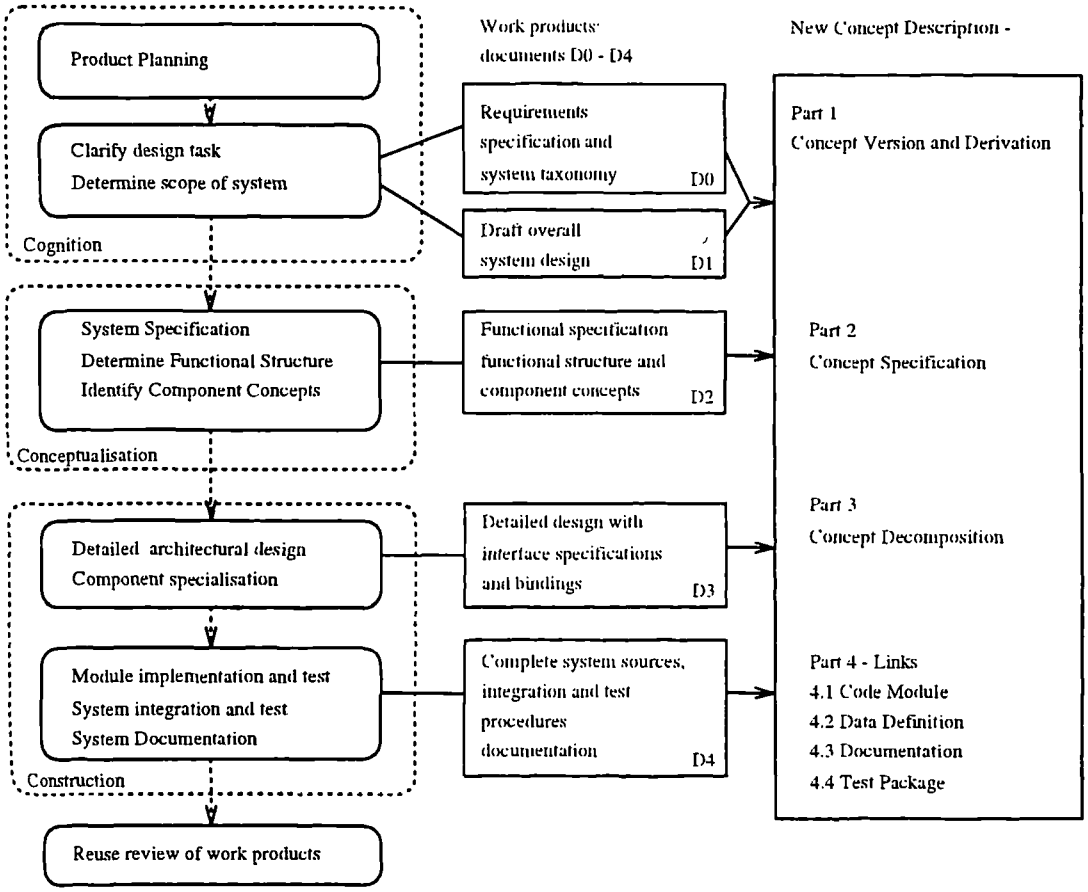


Figure 3.5: Software Life Cycle Work Processes and Work Products Related to the CDF

C reproduced with minor modifications from [26]. This appendix also includes an alternative concrete form with informal annotations detailing what is expected under each entry on the CDF. This was prepared for the benefit of engineers concerned with filling in CDFs during the Practitioner project. Appendix D contains a short example CDF which was filled in by the author during the domain analysis of control systems used in steel production. Here the CDF has been used to describe a software concept working from an account given in the literature rather than existing software documentation.

The semantic content of CDF's individual entries is undefined in the general case. In so far as any entry has a semantics, this comes from the language used to fill-in that entry. For example, a concept specification may be given using a particular specification language or it may be described in natural language. However, despite this lack of semantics, below a case will be made for considering the CDF as an interconnection language. The CDF will also be compared to related developments, including some interconnection languages, some of which like the CDF aim to support software reuse at a high level of abstraction.

Within an application domain, using the CDF to record design concepts, the potential to record networks of concepts exists. In this way, the CDF sets describing related concepts can be thought of as means of mapping out the conceptual space in a particular application domain given by the conceptual levels of abstraction at which they are described. This enables the requirement of describing the design from different viewpoints where these may be related as conceptual levels of abstraction in the application domain to be met through completing a number of related CDFs.

Figure 3.6 illustrates the CDF without any detailed contents, simply to show in general the way in which a concept description is given a structure through the CDF. Through the derivation relation, a means is provided to link generic concepts at one level of abstraction with specific instances. In Figures 3.7 and 3.8, a simplified graphical view of the CDFs required to describe an order processing system in a strip processing line is given. In these two figures, the view of all of the essential relations recorded using the CDF can be seen. These relations are as follows:

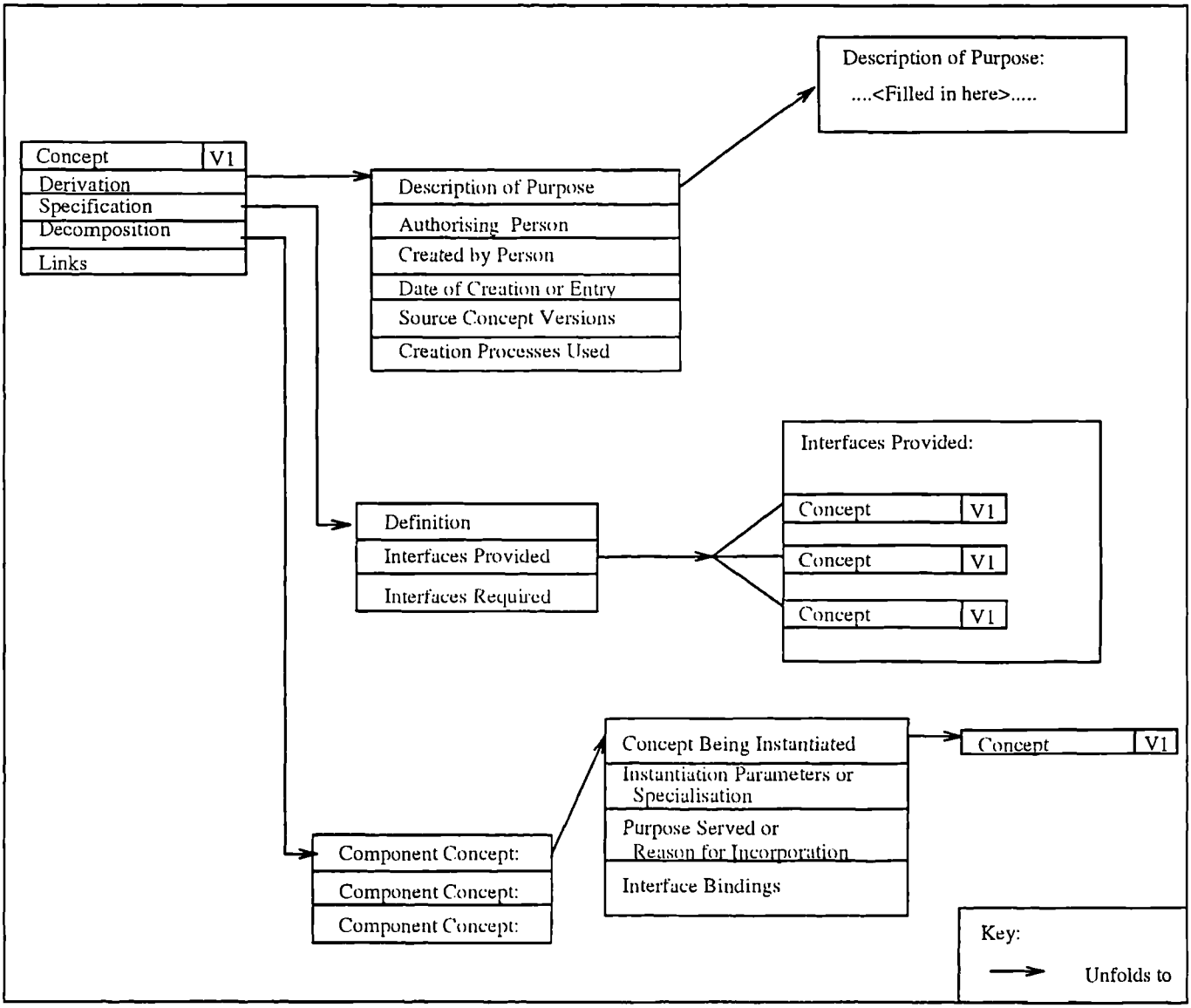


Figure 3.6: Progressive Unfolding of CDF Sections

1. version,
2. derivation,
3. decomposition, and
4. named links (serving as a catch-all for all the relations recorded in part 4 of the CDF).

The concept, Order Processing System, described graphically in Figure 3.7 is the source concept for the concept of Order Processing in a Strip Processing Line described in Figure 3.8. This can be seen by looking at the box underneath the label Derivation in Figure 3.8.

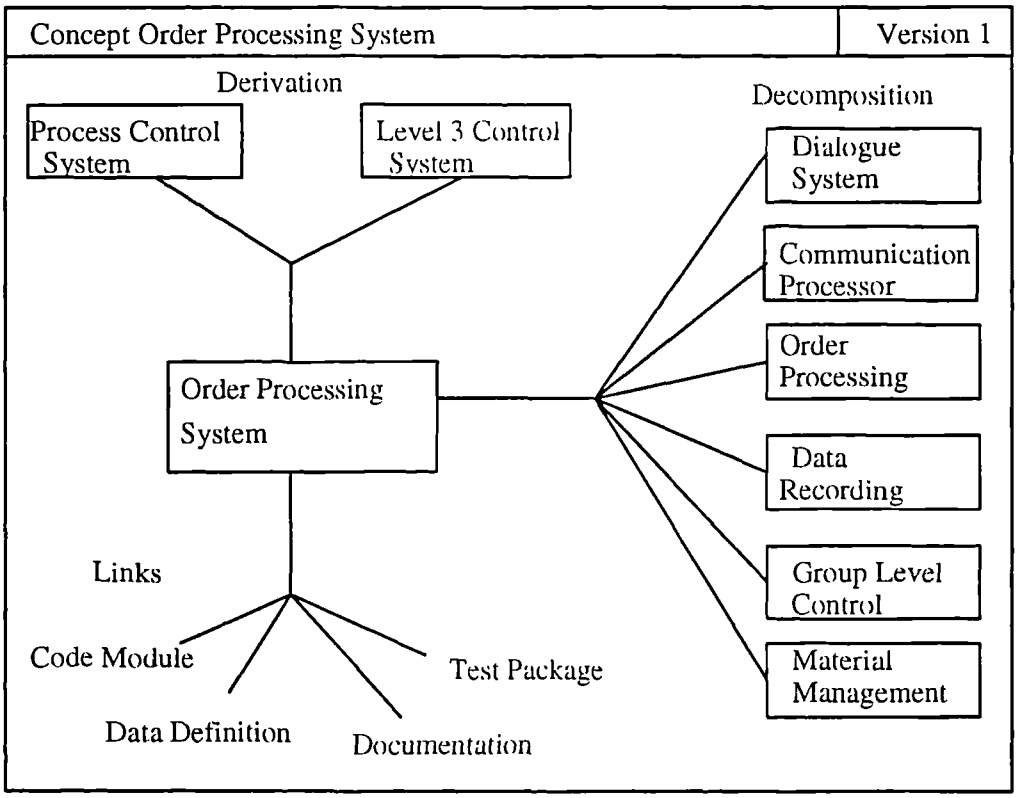


Figure 3.7: CDF for Order Processing System

These CDFs have been prepared by the author based on the study of earlier questionnaires filled in by engineers at ABB working from existing software documentation.

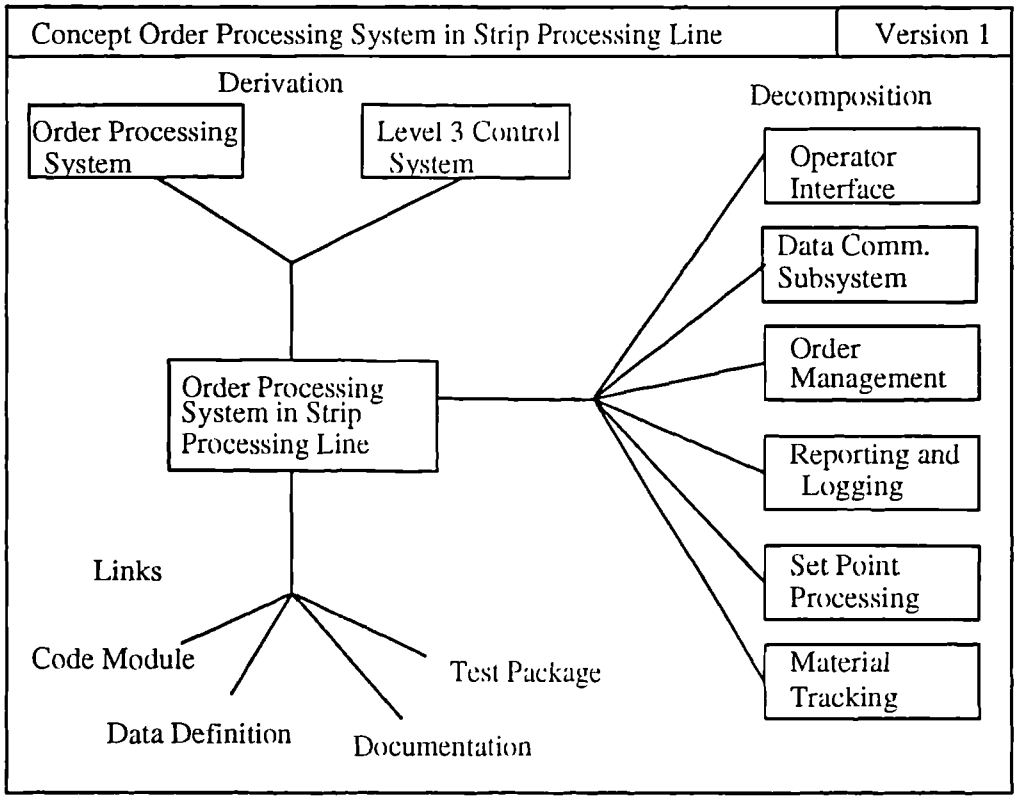


Figure 3.8: CDF for Order Processing in Strip Processing Line

3.5 The CDF and Related Developments

The CDF as presented here provides a means of describing the interconnection of software concepts. Using the CDF, a concept can be described in terms of the component concepts of which it is composed. The CDF also allows the interfaces provided and required by a concept to be identified; and the internal and external interface bindings to be described. In a set of CDFs, large hierarchically structured system concepts can be described through decomposition at successive levels. Using the derivation relation, a set of CDFs can also describe the derivation of specific concepts from generic concepts at different levels of abstraction.

A software concept is a more abstract notion than a module; and it has not been the intention to consolidate the design and construction process in a single description as with module interconnection languages. Its support for describing interconnections between software concepts has been detailed above. On this basis, the claim is made that the CDF can be classified as an interconnection language, although it is not

claimed that the CDF is a module interconnection language. To move the CDF as a language in this direction by giving more semantic content to its parts would defeat its purpose. During the phase of design known as conceptual design, such detail is not required. It is acknowledged that many aspects of existing interconnection languages influenced the development of the CDF.

From the work on LIL, the notions of horizontal and vertical composition were taken and support for these was established as a requirement. Unlike the LIL proposal within the CDF, these aspects of composition, i.e. its vertical and horizontal nature, are treated very loosely. The LIL proposal provides general parameterization mechanisms to support vertical composition that require much more explicit description of parameters than has been envisaged to support conceptual design using the CDF.

The way in which the LIL proposal aims to support both formal and informal axioms allowing whatever mixture seems most appropriate for a particular application has been more widely applied to arrive at the CDF's derivative approach to semantics, i.e. CDF entries derive their semantics from the language used to fill them in.

Developing the CDF further along the lines of the LIL proposal was ruled out as extensive reverse engineering of existing software would have been required to describe the software concepts realised in a LIL-like formalism.

From SySL, the distinction between interfaces provided and interfaces required, found in many module interconnection languages, has been taken over and used in the CDF where it is applied in general to any concept's interfaces. The CDF goes beyond SySL in allowing interfaces to be treated as concepts in their own right with associated CDF descriptions. Unlike SySL, interfaces have been further distinguished as external and internal; the external interfaces of a concept are those described at the top level in its specification while internal interfaces are those of its component parts.

The CDF bears some similarity to the architecture frameworks language of Henderson and Warboys. The middle part of the CDF with decomposition is equivalent to

the notation for architectural frameworks given by Henderson and Warboys; however, their development in the form of the Alf language constitutes a departure from the minimalist form of the CDF. Their work does not address the establishment of links with existing software documents which is a key element of the CDF.

Finally, the CDF as a canonical form for describing software concepts, both component concepts and architectural concepts can be compared with the proposals of Weber based on CEM descriptions to support integration of reusable software components. Here for purposes of comparison to the CDF, it is simply noted that the constituent parts of a CEM are as follows:

the export interface,

the body,

the import interface, and

the common parameters of the export and import interfaces

and that these correspond quite closely with the constituent parts of the CDF's notion of Concept Specification, i.e.

Interfaces Provided,

Function,

Interfaces Required, and

Generic Parameters

which are reordered here for ease of comparison.

In a wide ranging discussion, Weber advocates that industry move towards reuse-oriented software engineering with the canonical software component providing the conceptual basis. He goes further introducing the notion of component archetypes, where these are classes of reusable components with some commonality in their domain semantics. In many respects these resemble generic component concepts

Table 3.1: Comparison of CDF with Other Approaches

Requirement	LIL	SySL	AFL/Alfa	CEM	CDF
R1 Generic Architecture Derivation	Yes	Yes	No	Yes	Yes
R2 Generic Component Derivation	Yes	No	No	Yes	Yes
R3 Concept Compositions	Yes	Yes	Yes	Yes	Yes
R4 Interfacing Concepts	Yes	Yes	Yes	Yes	Yes
R5 Conceptual Viewpoints	No	No	No	No	Yes
R6 Explicit Links to SLC products	Possible	Partial	No	Unclear	Yes

described by the CDF. Their intended use is to refer to a proper placement of a particular reusable component into a classification schema. From the text, it is not quite clear how archetypes will provide a classification schema in practice. In his example from electrical engineering, standard component types are simply enumerated.

In addition, he introduces the notion of canonic software architecture and architecture archetypes; these bear a resemblance to specific architectural concepts and generic architectural concepts described using the CDF. As examples of architecture archetypes, Weber mentions the proposed Systems Application Architecture (SAA) from IBM and the European Software Factory's reference framework.

Weber's proposals are rather visionary and have yet to be realised in a practical application. Yet in these proposals there is enough resemblance between his canonic forms and the CDF to confirm the approach taken to describing software concepts using the CDF. His emphasis on formally defining the integration of software components as a means of describing the embedding of prefabricated software parts into a system environment goes beyond conceptual design towards the phase of embodiment in design, i.e. the actual construction of software systems, an area which is outside of the realm of the work reported here.

Table 3.1 compares the CDF with these other developments with respect to the six requirements identified earlier in this chapter.

Finally, consideration is given to comparing the CDF to faceted classification as a means of representing software concepts.

3.5.1 Relation of CDFs to Faceted Classification

Above in earlier sections, the CDF has been compared to various languages notably those identified as interconnection languages as well as recent developments to aid in the description of reusable software at a high level of abstraction. The principle advantage of the CDF over these developments was identified as its accommodation for descriptions at various levels of conceptual abstraction. In Appendix A, the approach within the Practitioner Project of combining concept description based on existing documents with a faceted thesaurus is explained in detail. However, faceted classification might be sufficient in itself for the purpose of describing reusable components. In what follows, consideration will be given to the question whether or not faceted classification on its own provides an alternative to the CDF used in conjunction with a faceted thesaurus to record the results of domain analysis. It could be that restructuring of software documents into the CDF is not required if the classification system provided by using facets itself describes the domain concepts and their relations adequately. Here it is argued that faceted classification on its own is insufficient for this purpose.

Taking the concepts described above in terms of a set of CDFs, two hierarchies of relations can be obtained using the derived and decomposed relations. The CDFs also describe richer relationships between concepts via versions and interfaces but for the time being these will be ignored.

Applying faceted classification to the software concepts described using the approach described in Prieto-Diaz's account given in [125], does not result in any ordering on the classified items. The class of an individual item is determined by selecting the term under each facet which best characterises that item. For software classification, a partial listing of terms and facets can be found in the above cited paper by Prieto-Diaz.

An ordering can be imposed by constructing a faceted thesaurus as was done in the Practitioner project and has been illustrated in Appendix A, but the terms in the thesaurus must correspond to concept descriptors to have the result of imposing

an order on the concepts in the domain. In addition, expressing the relationships of derivation and decomposition among concepts is problematic. Here is the crux of the problem. The thesaurus relation of narrow term to broader term is not one of part to whole expressed by the decomposition relation, nor does it correspond exactly to that of generic to specific expressed by the *derived from* relation.

There may appear to be analogies between the thesaurus relation BT (broader than) and the CDF relation *derived from*, the thesaurus relation NT (narrower than) and the CDF relation *decomposed into*, and the thesaurus relation UF (use for), i.e. synonym for, with the CDF relation *version of*. Further consideration shows that the analogies are superficial. The terms compared using BT and NT are all of the same type, i.e. fall under the same facet. Whereas the *derived from* relation applied to concepts is that relating a general concept to a specific instance concept, not objects of the same type. The relation *decomposed into* relates a whole to its part; these also are of different types. The term for a part is not narrower that than for a whole, it describes an object of a different type. For example, the term, code generator, is not a narrower term for the term, compiler, even though a compiler's decomposition may include a code generator. Nor is the term, compiler, a broader term for the BLISS/11 Compiler; in fact proper names are not conventionally entered in thesauri.

An alternative to the thesaurus order can be obtained by employing the method of conceptual graphs described by Prieto-Diaz, but this involves ordering the terms under a facet by placing them on the nodes of a directed-acyclic-graph and placing weightings on the edges. In the example shown, the graph employs the convention of distinguishing some terms as supertypes; these appear at the top of the graph with other terms connected as leave nodes. The graph shown thus has a depth of 3 levels (root - supertypes - nodeterms). A workable example applicable to software classification is lacking. This approach is equivalent to ordering the terms under a facet using the thesaurus relations using BT, NT and UF at one level and the weightings impose a further ordering on the narrower terms which using standard thesaurus relations could be achieved by appending the level number to the thesaurus relation, NT, and exploiting the fact that if A is a narrower term of B, then B is a

broader term of A. The latter convention allows a deeper ordering of terms if the thesaurus is then interpreted a directed graph.

A further problem is that the concept categories are usually synthesized from the term entries under the given facets. That is the terms used in faceted classification are not necessarily themselves denotations for concepts, so any relationships defined amongst these terms are not necessarily reflections of relations between concepts *per se*. It is certainly unclear to the author how the weightings between terms can be translated into a classification system with an interpretation of ordered classes based on conceptual graphs of individual terms out of which these class headings have been synthesized.

The use of CDF allows specific relations identified between software concepts to be clearly expressed. A categorisation of concepts can be obtained using the faceted thesaurus as explained in Appendix A, but it will not be able to show the same relations among concepts as a set of CDFs. The faceted thesaurus in the PRESS in practice has not been employed for this purpose. It is instead used in understanding the domain terminology and for indexing of CDFs to support their subsequent retrieval.

A further disadvantage of the faceted classification approach on its own is that it requires each item be given a single classification based on selection of a set of terms from the relevant facets used. This is in line with a rule of standard classification practice which is to place each item in a single category. As such, the approach results a consistent placement of similar items. Under the CDF relations, such consistency is not required nor expected, the same concept may be be variously derived and may have a number of quite different decompositions. Maintaining consistency between sets on CDFs is not a goal of the work as described here as in many domains, various conceptual approaches are used in practice.

3.6 Conclusions

This chapter has analysed the requirements for a language to support software concept reuse and developed the Concept Description Form, the CDF, out of earlier work to meet these requirements. It has been shown how this development is potentially adequate for recording software concept descriptions at a sufficient level of detail that supports the possibility of their reuse in conceptual design. Its support for describing both horizontal and vertical compositions of software concepts allows it to be classified as an interconnection language. This underlies its use as a basis for the design frameworks approach to reusability which will be proposed in subsequent chapters, where in the practical application of the CDF, a design framework will be developed and populated using sets of CDFs.

Because of the abstract nature of software concepts employed in conceptual design, the CDF has not been developed along more formal lines particularly with respect to specification of interfaces between concepts. It remains simply a minimal form deriving semantics from its contents. Here the approach taken is confirmed by that of Henderson and Warboys and classic Construction Theory [110]. The horizontal and vertical compositions of software concepts described by related sets of CDFs are intended simply to sketch out designs in the conceptual space of an application domain. The next chapter provides further illustrative examples of the CDF and outlines its usage through a small scale application.

The CDF has been developed specifically to support the description of software concepts as realised in existing software with the intention that such descriptions obtained using the CDF will provide a basis for conceptual understanding and allow subsequent reuse of these software concepts in future designs. The originality of the CDF lies in its support for recording descriptions of software concepts at various levels of abstraction, for recording the relations among software concepts, and linking software concepts with existing software lifecycle documentation if relevant. Through studying relations among a set of software concepts, the potential reuser is provided with a contextual basis for understanding the software concept and for its

subsequent reuse. None of the existing work reviewed above has been concerned to address this aspect of software concept description. In the subsequent chapters, the applications of the CDF in practice will illustrate the nature of the sort of conceptual understanding that sets of CDFs can, in fact, be used to record, and it will be demonstrated how such sets of CDFs can be used to support both design for reuse and design with reuse.

Chapter 4

A Simple Example Applying the Concept Description Form

This chapter consists of a simple application of the CDF for illustrative purposes. The domain to be studied is one which is well understood and documented and few problems have been encountered in carrying out the domain analysis. In conclusion, consideration is given to how the CDFs obtained could be used to support reuse.

There are two principal questions which the work described in this chapter sets out to answer; they are as follows:

1. Can the CDF be used to record the results of analysing a well understood and documented domain where the goal is not only to describe individual software concepts but more importantly to describe the relations amongst the software concepts in the domain?

2. How useful are the CDFs obtained as a result of the work in actually supporting design for reuse and design with reuse?

The application to be described here can be thought of as constituting a control

experiment. If the work undertaken in applying the CDF in this simple case doesn't allow the above questions to be answered positively then the practical utility of the CDF is doubtful.

4.1 Introduction to the Domain

The software concepts to be analysed are those employed in compiler design. The multi-phase compiler is one of the few architectural concepts that has achieved widespread acceptance and in which every software engineer is expected to have been trained [114]. Following Perry and Wolf, in this simple application of the CDF, this familiarity is assumed. Their classic multi-phase compiler model distinguishes five phases: lexical analysis, syntactic analysis, semantic analysis, optimisation and code generation. The phases progressively transform the source program input into the target program output as illustrated in the figure below. Generally, a compiler

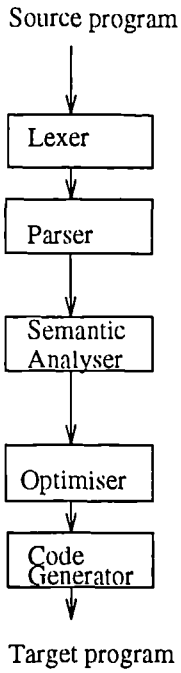


Figure 4.1: The phases of a sequential compiler

built using this model will employ the following component concepts corresponding to these five phases:

1. lexical analyser or lexer,
2. syntactic analyser or parser,
3. semantic analyser,
4. optimiser, and
5. code generator.

The way in which the components are organised together in a particular compiler depends on the architectural concept employed in its construction. For example, one which will be described here is a simple sequential compiler where each phase of the compilation process is carried out in a fixed sequence and the components are connected in sequence with the output of one becoming the input of the next. This overview lacks details. For example, it doesn't say if the concept of a sequential compiler is derived from a more general concept, nor does it say how the component concepts are connected together and it omits to say what the principal interfaces of the concept are.

For these levels of detail, a more detailed analysis of the relevant software concepts is needed, and here for illustrative purposes, the CDF will be used to record the results. The resultant completed CDF set will then be available for consultation and provide such detailed information to potential reusers of the concepts. In the example that follows, application of the CDF is considered stage by stage. Initially the application of the CDF is considered from the standpoint of relevant domain analysis necessary to identify principal sources of reference about the concepts, and finally consideration is given to showing how the various accounts of the concepts found in the reference source can be recorded using the CDF.

4.2 Background and Overview of Domain Analysis

The broad steps in Domain Analysis have been outlined by Prieto-Diaz [123] as follows:

1. prepare domain information,
2. analyze domain,
3. produce reusable workproducts.

Below consideration is given to each of these steps in turn.

4.2.1 Step 1 - Prepare Domain Information

In the first step, it is relevant to determine the domain, its principal sources of knowledge and its requirements for reuse support.

There are several textbooks which describe the software concepts relevant to the construction of a sequential compiler. One classic reference is that by Aho, Sethi and Ullman: Compilers Principles, Techniques and Tools [4]; this book itself is descendant of an earlier work by Aho and Ullman [5]. The five phase sequential compiler presented above is a simplification of the multi-phase compiler presented by Aho, Sethi and Ullman in their introductory chapter. The body of the textbook considers how each phase is realised in principle with specific reference to an actual compiler that is presented. Chapter 12 of their book describes how the components corresponding to the various phases have been connected together in various specific existing compilers. Thus this text provides all the information required to construct a set of CDFs covering the principal software concepts used in compiler design. As well as enabling general concepts to be described, the information on specific

compilers enables the specific concepts derived from these more general concepts to be described.

A thorough-going analysis of the software concepts relevant to the design of compilers can be obtained by studying this book and follow-up studies on the specific compilers would be needed to supplement the brief accounts found in the main text. Here it is not considered necessary to carry out such a study in depth. In order to illustrate the use of the CDF only a few concepts will be described, and in order to focus on the CDF, the concept descriptions themselves have been kept quite short. To simplify the presentation, the CDFs will be presented using the graphical form proposed in [31] and already introduced in Chapter 3.

An important aspect of carrying out a domain analysis in order to support reuse would normally be to establish the needs for such reusable design concepts to support reuse in a the domain of interest. Here this study is somewhat artificial because as remarked earlier, the design concepts used in compiler construction are already well known to the experienced compiler writer and have been described in a number of textbooks. However, a novice compiler designer might still find it useful to be able to consult a set of CDFs to gain an overview of the material presented in a standard reference work such as Aho, Sethi and Ullman. Such an overview could also serve as an introduction to students of Software Engineering.

4.2.2 Step 2 - Analyze Domain

The second step in Domain Analysis involves studying the sources of knowledge identified in light of the requirements in order to identify the concepts in terms of which the domain-specific knowledge is expressed. In this study where the domain-specific knowledge is concerned with compiler construction, the object has been to identify the generic models employed in compiler construction and generic component concepts found in common decompositions. These are directly described in the literature. If this were not the case, then initial studies of specific software systems in order to abstract the design concepts would be necessary at this step in order to

determine conceptual basis of the domain.

CDFs will be used to record the concepts and their relations found as a result of this second step. At this stage, the CDFs need not contain detailed specifications or links to realizations such as code although in some cases, it may be possible to establish such links at this stage. If specific existing systems are studied, then the concepts analyzed will obviously have known realizations.

The Practitioner Project augmented this second step with studies of the terminologies used in the domain to determine the domain vocabularies used to describe concepts from various viewpoints. Such a study has not been undertaken here as the main source was not available in machine-readable form, and in any case preliminary reading determined that it does employ a reasonably consistent vocabulary.

4.2.3 Step 3 - Produce Reusable Workproducts

The final step involves filling in the details necessary to support reuse of the concepts identified. It consists of developing the reusable workproducts required by the reusers in the domain. Here this involves adding information to the CDFs to the level needed; this has been determined earlier by the reuse requirements that the CDFs are primarily for illustrative and perhaps educational purposes.

4.3 Details of the Software Concepts Studied

The principal software concepts described in the Aho, Sethi and Ullman book can be summarised as follows:

1. compiler,

2. simple one-pass compiler,

3. lexical analyzer,

4. parser,

5. simple type checker,

6. run-time support package, and

7. code generator.

These concepts have been identified by studying the details found in the book's contents pages and initial chapters. The book also mentions "cousins" of compilers such as preprocessors, assemblers, loaders and link-editors as well as compiler-construction tools such as parser generators, scanner generators, syntax-directed translation engines, automatic code generators, and data-flow engines. These "cousin" concepts are used when the book describes the "context" of a compiler within a language processing system which may consist of preprocessors, a compiler, an assembler, a loader and link-editor. Brief descriptions of the following specific compilers can also be found in the book:

1. EQN, a preprocessor for typesetting mathematics,

2. Pascal compilers,

3. C compilers,

4. the BLISS/11 compiler, and

5. a Modula-2 optimizing compiler.

Using the relations *derived from, version of* and *decomposed into* which the CDF supports, the following potential candidates for recording have been identified and can be related as shown in the Figure 4.2 employing the graphical view of a concept introduced in Chapter 3.

Here there are six CDFs; reading from left to right and starting at the top of the figure, these describe the following concepts:

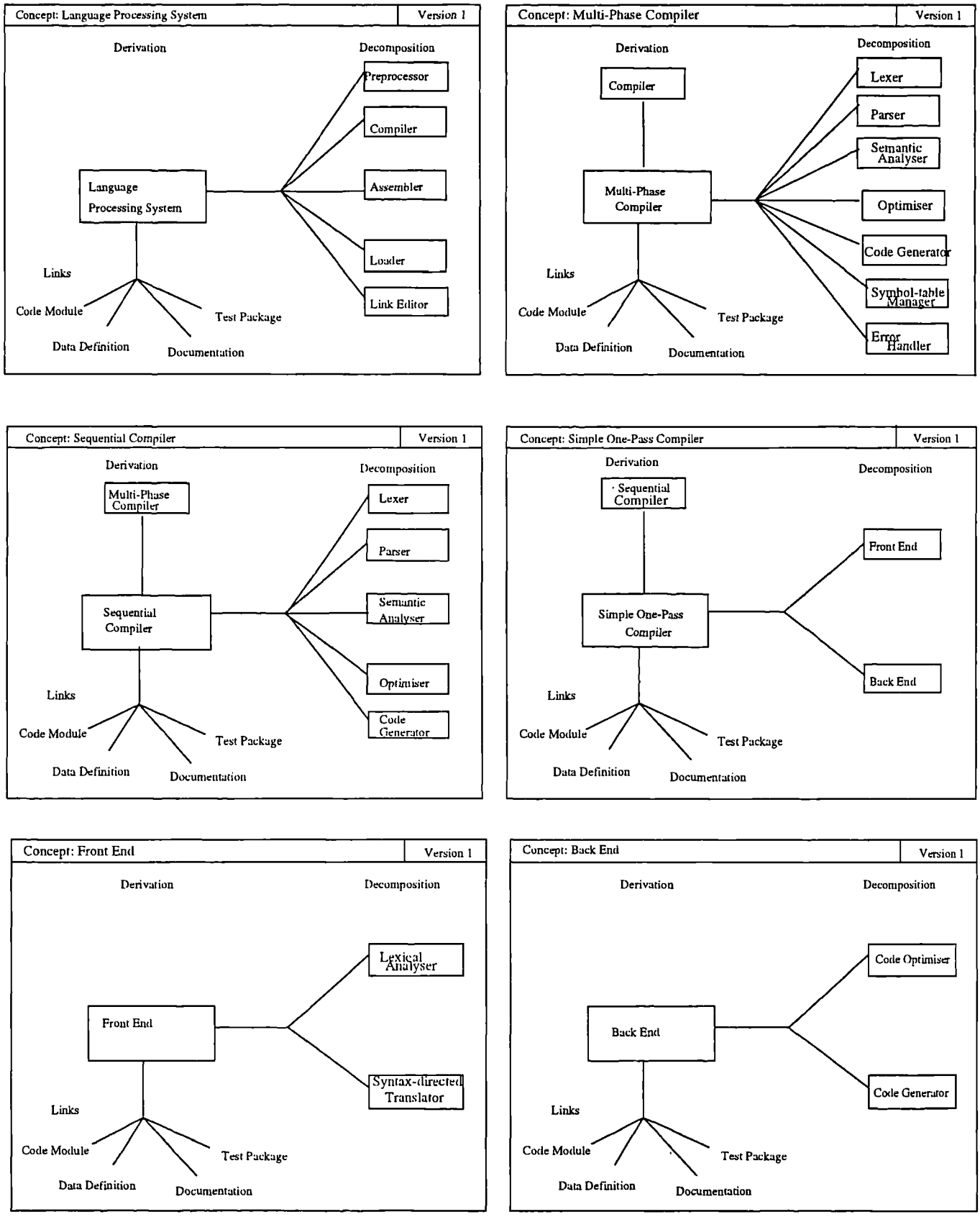


Figure 4.2: Domain Specific Concepts in Compiler Construction

1. Language Processing System,
2. Multi-Pass Compiler,
3. Sequential Compiler,
4. Simple One-Pass Compiler,
5. Front End, and
6. Back End.

The first concept, Language Processing System, sets the overall context of a compiler. The next concept, Multi-Phase Compiler, encapsulates the basic architectural concept with which most software engineers are familiar. The third concept is a specialisation in the form of a sequential compiler, and the fourth is a specialisation of the third. The fifth and sixth are concepts used in the construction of a simple one-pass compiler which is a special case of a sequential compiler. Note at this point, the concept descriptions are not based on specific existing compiler documentation. These general system models are the result of step two of the domain analysis process outlined above.

With respect to the detailed description of reusable work products, here the subsequent recording of details is confined to considering a specific concept, that of the BLISS/11 compiler shown here in Figure 4.3. As the book explains, this concept is derived from the concept of a simple one-pass compiler which has already been recorded and shown in Figure 4.2. Note although the BLISS/11 compiler is described by Aho, Sethi and Ullman as a single pass compiler; the mapping between the five components of this concept and those of the Simple One-Pass Compiler concept given in Figure 4.2 are not obvious. A more detailed description of these components is given in the book, and this together with the more detailed description by the compiler's authors would need to be recorded to clarify the derivation relationship between the BLISS/11 Compiler and a Simple One-Pass Compiler.

Figure 4.4 shows some of the recorded detail taken from the account of the BLISS/11 Compiler given in Chapter 12 of Aho, Sethi and Ullman's book. The concept's two

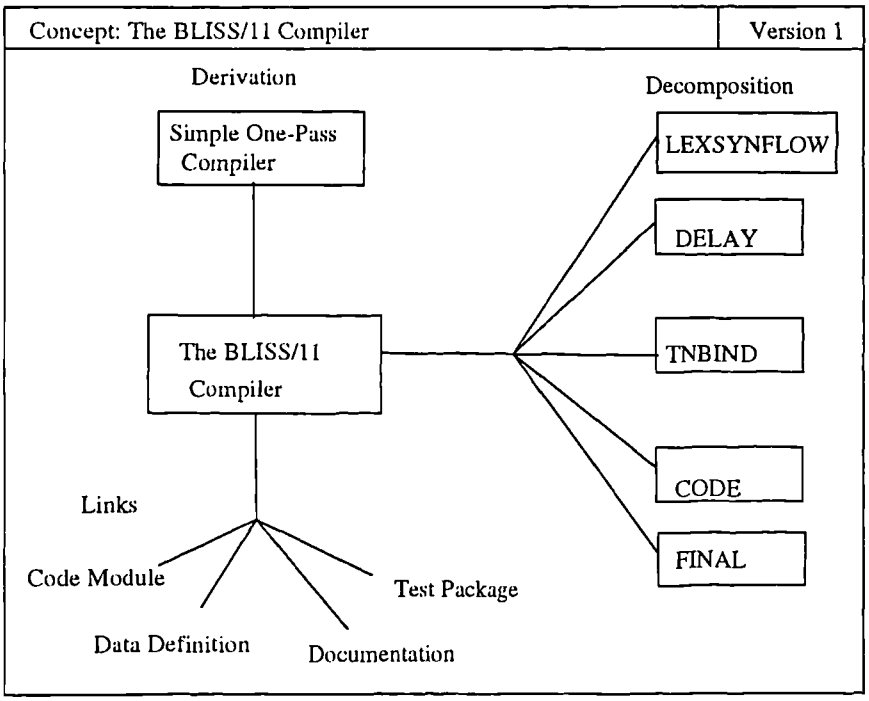


Figure 4.3: Overview of the concept of the BLISS/11 Compiler

main interfaces have been revealed by unfolding the the CDF's Specification section and then unfolding the two sections: Interfaces Provided and Interfaces Required. Unfolding the Links section followed by unfolding the Documentation section reveals the two main references used in obtaining the detailed description of this concept.

As this application of the CDF is primarily for illustrative purposes a more detailed filling in of the CDFs identified above will not be undertaken here.

4.4 Consideration of Design-with-Reuse using Compiler Concepts

In order to make the above set of CDFs available to reusers, they could be installed in the PRESS. This would involve the domain engineer selecting appropriate indexing terms based on text analysis. Once installed, the potential reuser with a compiler development task could retrieve these concepts through a search featuring any of the indexing terms chosen. Once at least one of the compiler CDFs has been retrieved,

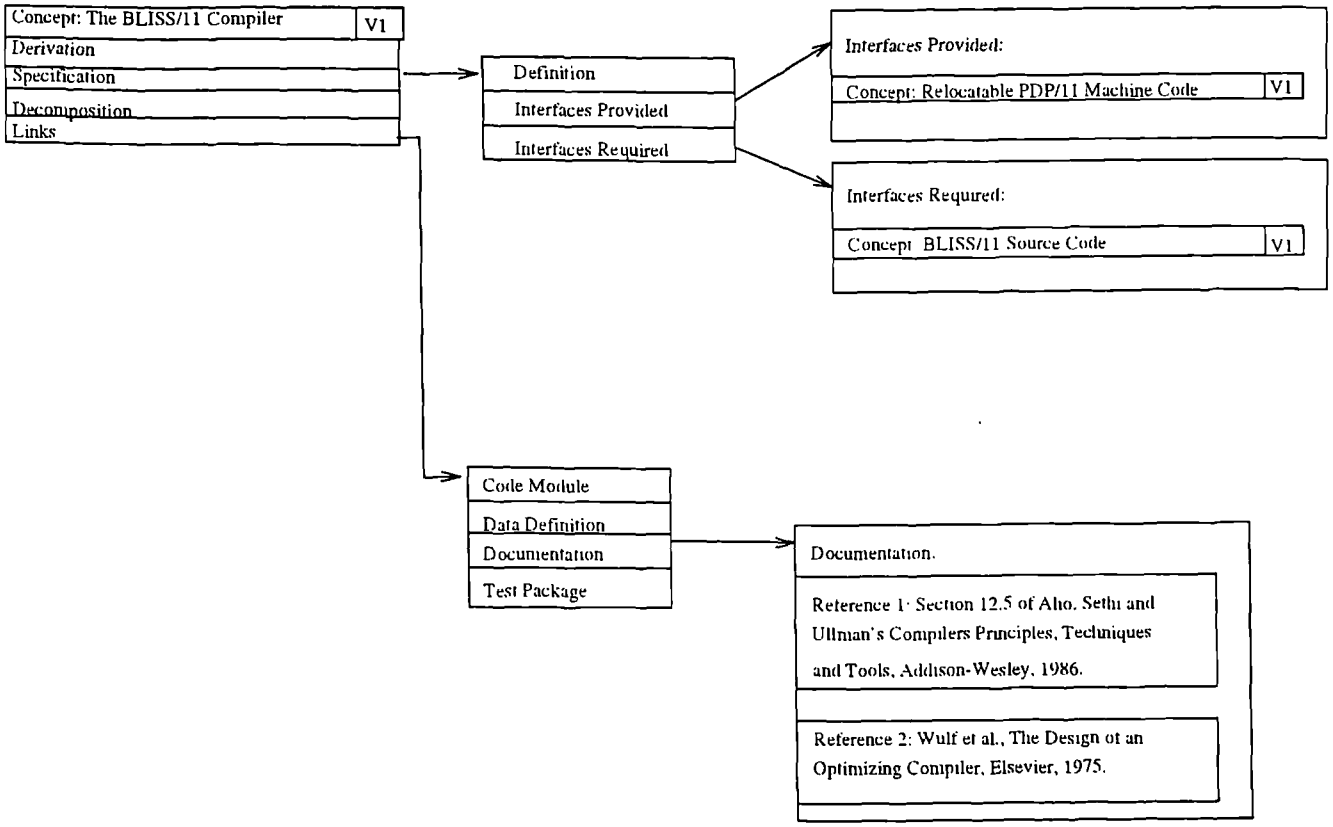


Figure 4.4: The BLISS/11 Compiler CDF Unfolded

the potential reuser will access to all of its related concepts.

For example, for a given concept, all of its related concepts could be displayed. If a specific concept has been selected, the reuser has the possibility of considering more general concepts which may not have occurred to the reuser as potential solutions by inspecting the concept's derivation. In the case of the concept *Simple One-Pass Compiler*, the derivation recorded in the CDFs is as follows:

> Simple One-Pass Compiler *is derived from*
> Sequential Compiler *is derived from*
> Multi-Phase Compiler *is derived from*
> Compiler.

The reuser could trace this derivation by systematically inspecting the concepts given under the Derivation heading in the graphical form of the CDF. After inspecting these and reviewing the more detailed descriptions, the reuser may decide that a Multi-Phase Compiler with a number of passes is more appropriate.

On the other hand, if a more general concept has been selected initially, the reuser might find through inspection of its related concepts a specific concept which may be exactly what is needed.

Through the other relations recorded, an inspection of the component concepts could be made and the relations between the main concepts' interfaces and the component concept interfaces could be inspected. In the case where the reuser wishes, say, to employ an existing parser, it would be possible to determine the nature of the interface required to make this substitution within the recorded compiler design.

4.5 Conclusions

This chapter has presented a simple application of the CDF in order to illustrate its usage to structure descriptions of software concepts. It has related the usage of the CDF to the process of domain analysis and shown how it can be used to record relations amongst domain concepts which can be then be described in more detail as the domain analysis progresses. In this way, the CDF provides a link between the analysing of the domain and the development of reusable work products as the CDF can be used for recording the results in both steps.

The domain studied, that of compilers, is one which is already well understood and documented; thus the description of concepts was able to progress from the general to the specific with the more general concept descriptions providing the framework for those of the more specific concepts. In the large scale example described in Chapter 5, specific concept descriptions were developed before any general descriptions, but as more general descriptions came to be known, these provided the basis for a more systematic approach to applying the CDF.

The main lessons learnt from this simple application are that sets of CDFs can be used to record the results of domain analysis in a well understood and documented domain. Concepts realised in both general models of compilers and in specific compilers have been described and related through the CDF. More work would be required to fill in the additional details and to record further software concepts from this domain, but the general software concepts already described provide the starting point for further design for reuse in this domain. The concepts described here could provide the basis of an initial introduction to compiler design and assist the novice designer in designing with reuse.

Chapter 5

A Large-scale Application to Support Reuse of Software Concepts in the Domain of Steel Production

5.1 Introduction

This chapter describes a large-scale application of the CDF in order to support software concept reuse in an industrial setting. The industrial setting chosen is that of software development in the domain of Steel Production. The opportunity to study software concepts in the steel domain came initially from the close link that this research has with the Practitioner project. The author has chosen to continue working within this domain because it is typical of many application domains where the benefits of software reuse are recognised, but software engineers working within the domain find themselves (to quote Jones [86]):

constantly faced with reinventing concepts which should be available from standard references.

This domain is also of particular relevance to the work on supporting concept reuse at a level of abstraction higher than code because it is a domain in which it has been recognised that design concepts rather than code have the most reuse potential. The studies of the Association of Iron and Steel Engineers support this emphasis. The third reason for developing the research within the steel domain arose when the author was able to extend the earlier domain analysis on control systems used in steel production with field studies of existing software actually in use at a steel mill. This allowed the author to confirm that concepts abstracted from existing software could be related to those from earlier domain studies and also gave the author valuable insights in the way in which software is typically developed in the steel domain. Finally designers of systems in the steel domain employ general concepts from control systems, so there were good sources of background concepts on which to draw in the domain studies.

There are three main aims of this application study using the CDF in the steel domain:

1. to apply the CDF recording the results of studies of both generic and specific software concepts in the steel domain (see the first study reported below),
2. to apply the CDF recording the results of studying specific software concepts abstracted from an operational system (the second study),
3. to apply the CDF to record the results of studying the more general concepts which are relevant to control systems in general (the final study below).

The main part of the work has been a domain analysis of software concepts used in the design of hierarchical control system software found in steel mills. A further part of the work has consisted of applying the domain analysis approach outlined in earlier chapters in a field study of process control software made at the Peine-Salzgitter

Steel Works. This work was undertaken to determine the utility of having a very general understanding of the domain to guide the identification and description of the concepts used in the construction of a particular system. The particular system studied was the system for calculating draft rolling schedules for controlling a tandem mill in cold working. In the final domain study, further work on the development of a design framework for underpining design reuse in this domain is undertaken.

The chapter first gives some general background to the domain, and describes the requirements for designing for software reuse and the ground work undertaken in its development using the CDF, presents the domain analysis results and the results of the work undertaken to further populate the Practitioner Reuse Support System (PRESS) with sets of CDFs from the domain of steel production control systems including those from software concept field studies carried out at the Peine-Salzgitter Steel Works. Finally, the chapter discusses the potential reuse of the software concepts described by presenting a demonstration based on consultations with domain experts and studies of how systems in this domain are typically designed.

5.2 Background and Overview of Domain Analysis Studies

5.2.1 Introduction to the Domain and Scope of Studies

The primary source for gaining an understanding of the domain of steel production used in these studies has been the report, *Tasks and Functional Specifications of the Steel Plant Hierarchical Control System* [169] (hereafter referred to as the Williams report). This report documents the results of over ten years research in this domain by the Purdue Laboratory for Applied Industrial Control (PLAIC). Their model of hierarchical control (derived from the work of Mesarovic, see [100]) has been acknowledged as a concept endorsed (in some form) by most practitioners in the domain and recognised as a concept that has brought a measure of order to the

steel industry's complex systems [46]. The overall model of control developed in the Williams report is based on a four level model. This model and its role in domain analysis will be discussed in more detail in this chapter.

An earlier work on the automation of control systems in metallurgy [49] provides an historical perspective on the development of control systems in steel production albeit mainly at the lower levels of control. However, even at this time, the potential for employing computer systems in the overall process of production control was recognised in theory; and the text contains a dataflow diagram illustrating flow of information in an order processing system in a rolling mill based on work carried out by the British Iron and Steel Association on computerising planning and accounting of a rolling mill. This diagram is very similar to that developed in the Practitioner project study of order processing in a strip processing line [89, 88]. And already in 1964, schemes of multilevel automation for the steel production process as a whole can be found to have been introduced (see for example, that by H.-J. Marx, cited in [74]).

Here in this introduction, only a brief overview of the steel production process and its associated areas of control is given, drawing on the Williams report. This report places its emphasis on process control in Steel Production; for a more comprehensive account covering all aspects of steel production, the reader is referred to the so-called bible of the steel industry - The Making, Shaping and Treating of Steel, 10th Edition [160] - (this edition was written by the United States Steel Corporation personnel with assistance from the Association of Iron and Steel Engineers).

Steel Production is basically a three part process; first the steel is produced in a molten state either from iron ore or scrap steel, or a combination of these, and cast or formed into ingots, then it is hot rolled, and finally it may undergo cold working. The output of hot rolling, hot band steel or hot rolled sheet, is typically used for heavy metal applications such as the framework of an automobile, in steel framed buildings, in bed frames, etc; whereas the output of cold rolling, cold band steel, is found in car exteriors, computer frames, and casing for other consumer products such as washing machines, video recorders, microwaves, i.e. the so-called "white

goods" of manufacturing. A modern steel mill will have areas associated with each of the three basic processes; these are the Melt Shop or Melting Area, the Hot Roll Mill or Hot Rolling Area, and the Cold Roll Mill or Cold Working Area.

The Williams report describes a basic steel product mill producing hot and cold rolled sheet. In the introductory chapter, the basic mill is shown diagrammatically divided into areas based on processes (see Figure 1-2, page 1-11 of the Williams report), and an argument is made for maintaining the distinct areas in the control system design models (see Figures 1-5 and 1-6 on pages 1-14 and 1-15 of the Williams report). These figures are reproduced here in a simplified form (modified slightly to show the alternative architectures) as Figure 5.1 - Areas of a Basic Steel Mill - and Figure 5.2 - Structure of Control System Design Reflecting Areas.

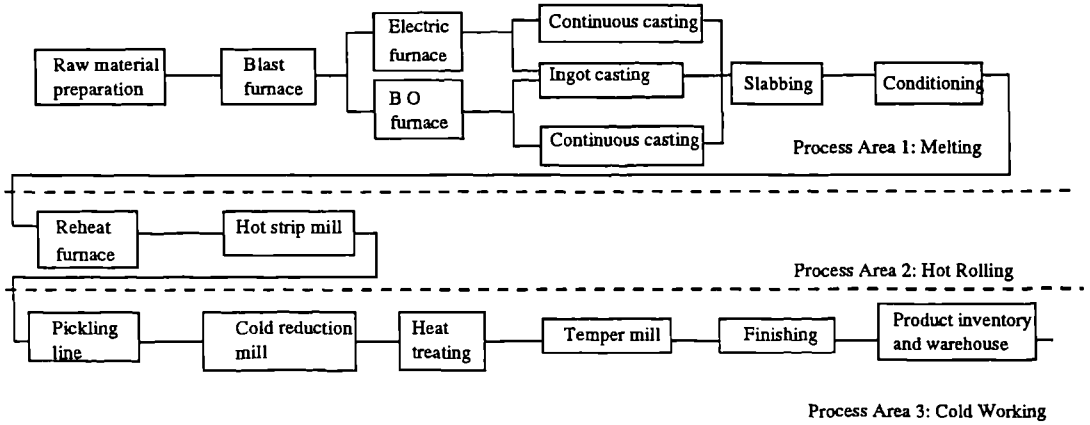


Figure 5.1: Areas of a Basic Steel Mill

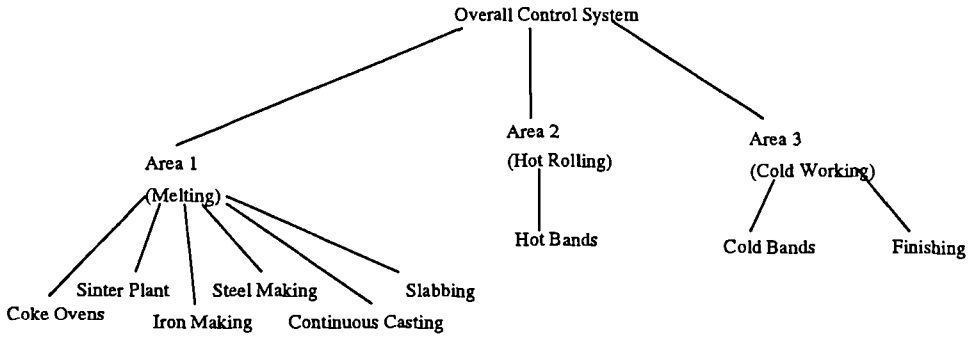


Figure 5.2: Structure of Control System Design Reflecting Areas

Although the Williams report describes the control systems typical of a "basic" steel mill, the same overall system structure can be found in Preston's popular articles on the Nucor Crawfordsville Project, a steel mill utilizing an experimental

casting machine developed by SMS Schloemann-Siemag AG [120, 121]. Here the main processes and plant areas are described in Figure 5.3 - Nucor Crawfordsville Project Main Processes and Plant Areas. This plant may be seen as a specialisation of the generic model given in Williams in that it employs the electric furnace option simply for melting scrap (such mills are known in the USA as minimills), and, because of its unique casting capability, only requires a "mini" hot rolling mill (in this case consisting of 4 roll stands instead of 6 or 7 found in a more conventional steel mill). It is clear from the accounts given that here as well the control systems are structured at the lower levels using the mill areas.

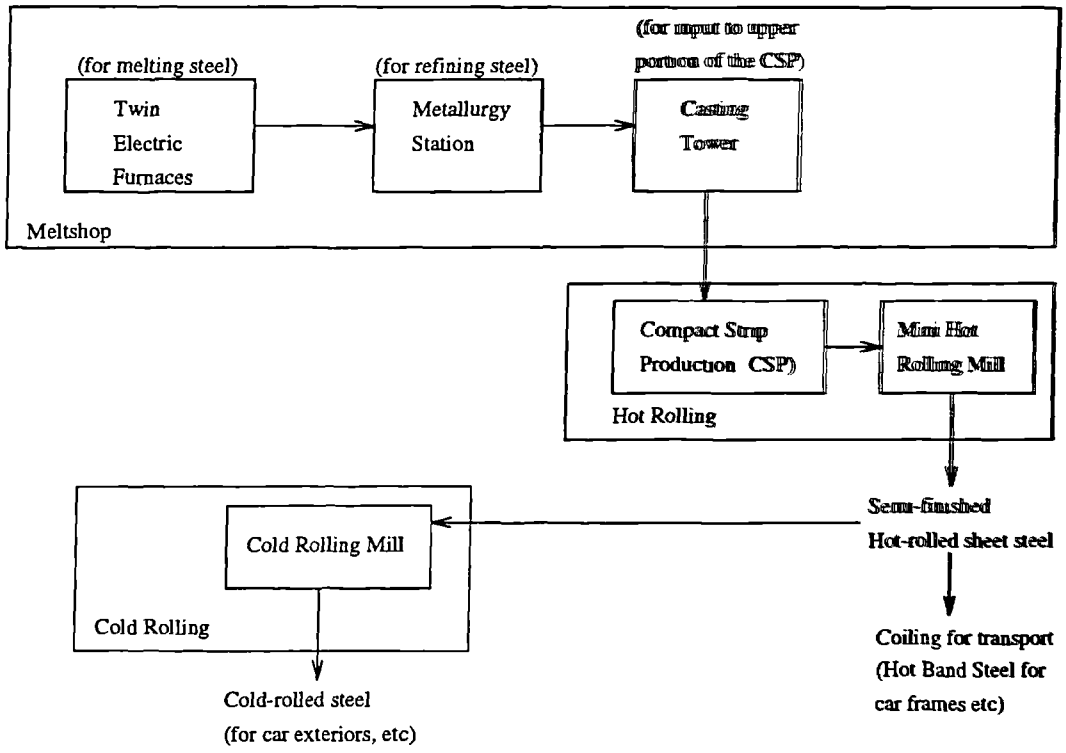


Figure 5.3: Nucor Crawfordsville Project Main Processes and Plant Areas

The steel mill whose software formed the subject of the supplementary field study, the steel mill at Salzgitter, falls somewhere in between the basic PLAIC example steel mill and the Nucor Crawfordsville Project (NCP). It produces a wider range of

products than the PLAIC basic steel mill, and unlike the NCP mill, it produces steel primarily from iron ore rather than scrap which means that control of its melting area and associated processes is considerably more complex. In the period from 1974 to 1985, the number of steel producers in Germany accounting for over 80% of crude steel fell from 20 independent companies to 7 groups of companies. The Salzgitter steel mill is owned by one of these groups, the Peine-Salzgitter group; its capitalisation in 1985 was just under half that of the Thyssen Steel, Germany's largest steel producer [102]. The study of software at Salzgitter was restricted to that used to control the tandem mill in the Cold Working area, and here the models given in the Williams report were supplemented by those found in a study specifically addressing tandem mill automation carried out by Bryant *et al* [40].

Before the research reported in this thesis was undertaken, published accounts of software concept reuse in the domain of steel production were not available although the need for these was recognised as discussed here in Section 5.2.2. There are no standard textbooks detailing the software concepts used in applications software supporting Steel Production in the way that these exist, say, on compiler construction principles as described in Chapter 4. The steel production domain is not unusual in this respect; as quoted in Chapter 2, Jones found that the reinvention of concepts which should be available from standard references was common in most major application software subfields [86].

Discussions with domain experts at ABB and Salzgitter highlighted one of the problems as being lack of a means of recording and preserving experience and knowledge gained in one project for use in another. As the development of software applications in steel production takes place often in the context of a larger engineering project which can take place over a number of years, the individuals on a particular project development are unlikely to remain constant. Therefore standardising software concepts employed over a number of projects has been difficult. Hence the need to carry out domain studies such as this to lay the ground for more systematic software concept reuse.

Also the system developments are likely to be piecemeal. For example, at Salzgitter,

software developments are usually related to a larger project of modernization of equipment in a specific area of the mill. The material made available for study from Salzgitter was concerned with improvements in the Cold Working area and involved development of new control software for the tandem mill. It is relevant to note that at the Salzgitter mill, the system documentation is not held centrally at the mill; it was found that all the tandem mill control system software documentation was actually kept in the central computer room of the Cold Working area. Nor was the documentation of a uniform standard as various contractors as well as employees of Salzgitter have been involved in its development.

From ABB, various sources of specific descriptions of software developments by the Metallurgy division were made available for study. The material was not in a standardised form; it was drawn primarily from projects concerned with the development of lower level control systems for steel mills. As the domain analysis progressed, it was possible to relate the ABB system models to the those given in the Williams report through a related model developed by Hoogovens as will be illustrated in the more detailed discussion on the results of the domain analysis and the role that these models played.

In summary, the scope of the domain studies developed during the course of the work. Orginally, it was based on concepts in ABB's existing software, then extended to the more generic concepts in the Williams report when this came to be known and could be related to the ABB material. In the field studies, further specific software concepts in existing software were studied as well as generalised versions of these concepts relating to tandem mill automation described by Bryant.

5.2.2 Requirements for Software Reuse in this Domain

Initially the requirements for carrying out a more detailed domain analysis to determine software concepts in steel production arose from the link that this research has had with the Practitioner project and its internal customer, ABB, specifically, the Metallurgy business division who develop control systems for the steel industry

throughout the world. ABB's requirements are related to those generally recognised within the international community of iron and steel engineering. Below the results of work by the Association of Iron and Steel Engineers (AISE) to improve software development in the industry are summarised. From these, the requirements for software reuse for steel production are derived. Finally the specific requirements for concept reuse that derive from the control system users in a steel mill are discussed; these were identified during the field studies.

With the increasing introduction of real-time information and control systems, the iron and steel industry through the AISE has recognised the problems of software development that are frequently recounted in more general accounts of the software crisis: high cost of software and the need to reduce this cost whilst preserving and improving quality and utility of software in the industry. To this end, the AISE software portability (later portability and productivity) project was established. This project consisted of three steps reported on in three separate reports. The account given here is based on reviews of these reports. Early on in their work, the project adopted a more general view of portable software as reusable software because in process control applications, their primary area of interest, transporting software without change was neither easily achievable nor always desirable. Thus the concept of reusability interpreted as transferring the underlying process technology and software design was deemed as a more realistic goal for the steel industry. (This conclusion can be found in the Review of Step 2 report [46]).

The AISE project also identified early on the developing practices of Software Engineering as having a crucial role to play in addressing the five most significant issues limiting portability (i.e. reusability) of process control software:

- lack of long-term planning for automation (resulting in islands of automation),
- benefits of reusable software need recognition to justify higher development costs,
- lack of standard plant operating practices,
- lack of standards for real-time data management,

- poor designs and subsequent need for system modification (also lack of standard specifications or guidelines for each major process unit).

Interestingly enough, this last issue was identified at a time (1984) when the PLAIC study was already well advanced with specifications of major process units' control and supervisory software systems in steel plant control systems.

Two aspects of Software Engineering were seen to be relevant to the Iron and Steel Industry's software development problems by the AISE:

- application of structuring principles to process control systems,
- application of software development methods appropriate to each phase of the lifecycle.

A layered model of process control software presented in the Review of Step 1 report [45] is directly relevant to the work populating and using the PRESS in the steel production domain. It is reproduced here in a simplified form in Figure 5.4 - Layered Model of Process Control Software System. This model incorporates the three essential elements found in systems studied during the domain analysis work:

1. automation of control (process technology software),
2. data acquisition (database software), and
3. communications (communications software, man/machine software and process I/O software).

The software solution to the communications requirements in the steel domain is fairly well standardised. In this area, the application standard software is partly based on international standards such as IEEE 802, standard products such as DECnet and industry standards such as HDN, the Hoogovens Data Network. Dialog with the operator may also be based on standard product such as DEC Forms Management System (FMS). However, this layered model is simply a starting point; it does

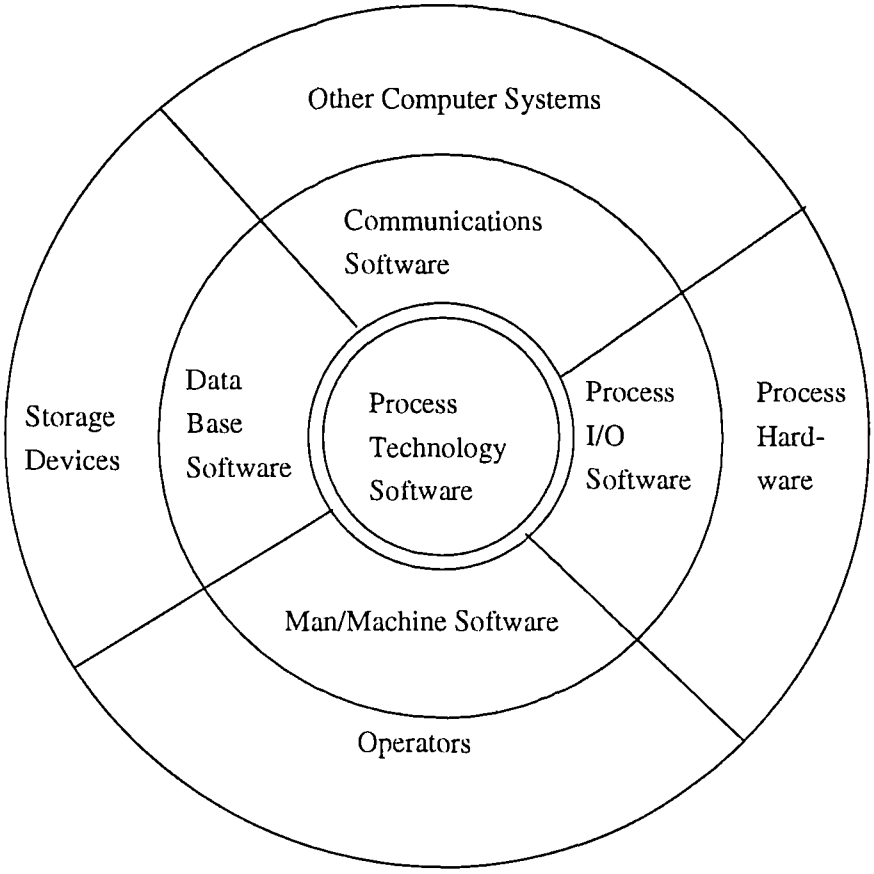


Figure 5.4: Layered Model of Process Control System

not address the levels of concepts within automation of control or communications; nor does it distinguish the classes of data objects acquired.

In the Review of Step 2 report (1985), it was recognised that there was a common ground provided by the parallel interests of the AISE and PLAIC investigations of control systems in the steel industry - work resulting in the Williams report from which the research reported in this thesis heavily draws. It is interesting to note that survey work in Step 2 confirmed the original analysis of obstacles to software reuse, and brought out two more:

- lack of a common language for process control applications, and
- lack of metrics for determining the benefits of reuse.

In their conclusions, the emphasis is on the need for standards, and with respect to standard software components, it is noted that the interconnection mechanisms receive little attention but are paramount to the success of this technique.

In the final Step 3 report reviewed in 1987 [47], European and Japanese practice was studied. Of the European companies studied, Hoogovens in the Netherlands is singled out as a notable success with a well organised approach to automation and long-term planning. The Japanese companies while well advanced in their application and use of automation had no common approach; all were addressing quality and reusability using different techniques and methods.

In their concluding remarks on the Step 3 report, software reusability research issues are noted as follows:

- no single methodology to support reusability among disparate application areas,
- lack of techniques to provide reliable means of storing and retrieving reusable software,

- lack of adequate documentation for reusable software, and of means to identify its function,
- lack of structuring principles applied in the design of software holding back reuse of functional designs;

and with respect to the steel industry:

- lack of management commitment for financial and technical support required to develop reusable software,
- the need for software library development and management,
- the need for educating software engineers in the steel industry about reuse methods and benefits;

and finally of paramount importance:

- need to impose and enforce reusability guidelines on outside software developers that supply the steel industry (i.e. software engineering requires engineering specifications in its contracts to ensure adherence to internal industry standards).

That these issues still hold for software systems in process control in the steel industry today is highlighted by Heidepreim's recent remarks:

> *One of the actual problems in automation of rolling mills today is the replacement of hardware and software of the process control computers that were installed in the 60s and 70s. It is impossible to save old software: control functions that are realised in software have to be rewritten. The reasons for this are the following: in many cases nobody (neither at the deliverer nor at the user) knows the old programs. Moreover the user wants to have new control functions to be implemented which could not*

have been foreseen 20 years ago, and it is difficult to combine these with the old program system. Finally Man-Machine-Communication (MMC) today is much better organised than in the past, and nobody wants to relinquish these modern facilities. - So, though software costs range about 80% of the total costs of renewal, it is indispensable to pay them.

[74].

Here a more optimistic view is taken with respect to at least saving old software concepts for reuse in design in this domain. That modernisation is inevitable cannot be denied, but the necessity of re-designing old systems from scratch need not exist if their conceptual basis is well understood. A design framework such as the one developed here is an attempt to provide such understanding, and to address the wider ranging requirements from the steel industry identified below.

The explicit requirements derived from the AISE project's reports that will be addressed by the domain studies to be related here are as follows:

- need for standards - particularly standard specifications or guidelines for each major process unit;
- need for application of structuring principles to process control systems, especially the application of structuring in designs to support reuse of functional designs;
- need for more attention to be given to interconnection of standard software components.

Additional more specific requirements for domain analysis to support concept reuse in this domain became apparent during the field studies at the Salzgitter Mill. These arise from the continuous need to improve models that underlie the control systems during their usage as part of evolutionary system development.

During the course of this study of the Salzgitter software, the Draft Schedule Calculation (SPB) system controlling the tandem mill was still subject to further development. A new model of rolling forces was being developed and tested within the SPB system. As with much industrial practice, where system documentation is manually maintained, new annotations were written on the existing documentation. One of the difficulties of keeping such system documentation up-to-date is lack of computer-supported version control and updating facilities. This is especially difficult if the definitive system documentation is a listing of the program in the supplier's assembly language as is the case for some of the SPB system. It was found also that the original documentation had sometimes been annotated by hand by readers, but there was no indication of the author, date or status of these remarks in the original documentation.

These problems apart, the adaption of the software by a new model is of interest because it makes clear that in complex process control, the development of software is in part experimental (see below under section 5.4.2, for a more developed discussion of the use of models in design in this domain). It goes without saying that software should implement the intended functions (and this must be tested, too), but the overall experiments are concerned with validating the models of control. In this context, poorly implemented software, or simply inadequately documented software, adds unnecessarily to the cognitive burdens placed on the system developers. Where users of a software system are likely to continue that system's evolution beyond its initial delivery, it is important for the provision of support for the conceptual understanding of the software to be made, hence the usefulness of an exercise such as the one described here. This point has also been made in the broader context of the AISE findings summarised above.

5.3 Specific Domain Analyses

The domain analysis carried out in the steel domain did not proceed in the same way as that recounted in Chapter 4. Here the distinction between the steps was not so

clear. Early studies using the original questionnaire carried out by ABB staff were based on specific domain concepts in existing software at ABB. These studies have all been made using the CDF to record both generic and specific software concepts. The studies divide into three as follows:

- first initial studies were made using the CDF to describe a variety of concepts without any established conceptual ordering;
- second having recognised the role of the more general system concepts, a field study of specific concepts was made to confirm the utility of the general system models which resulted from the first studies, and
- third a more detailed study of the foundations of these system concepts was made in order to consolidate the two earlier studies.

This final study sought to develop a design framework for software concepts used in the domain of steel production.

5.3.1 Initial Studies - First Applications of the CDF in Describing Software Concepts

In this section, experience using the CDF to describe software concepts is discussed. In the previous chapters, the concern was with addressing *what* aspects of software concepts should be described to best facilitate their reuse and illustrating how this could be achieved with the CDF. Now the concern to be addressed is *how* best to achieve such descriptions in actual practice. These earlier concerns were fundamental to supporting the design-with-reuse process. Before the PRESS could be effectively populated with software concepts, some consideration of the design-with-reuse process was necessary. The focus of this research has been to consider how best to make use of existing software descriptions, ranging from offers to build the software to academic reports on software, to derive software concept descriptions using the CDF.

A number of case studies of filling in CDFs are analysed. The process of filling in CDFs remains founded entirely on individual intellectual effort; it has not been possible to develop an automatic basis for transforming existing software documentation into concept descriptions realised using the CDF nor has any method of filling in CDFs amenable to computer assistance emerged other than the use of a standard text editor. Nevertheless it will be shown that a more systematic way of approaching the task of domain analysis using the CDF has emerged as a result of these studies, and that this constitutes some progress towards a better understanding of design-for-reuse. This case will be made in Chapter 6.

A study is made into how use of the CDF is supported by an appropriate system model; this study is based on experience gained from using the model of the steel plant hierarchical control system developed at Purdue University as a framework for relating the various designs studied. This model is described in the Williams report. The experience gained in filling in CDFs and using the model was distilled into the form of guidelines for future users of the CDF whether they be describing existing software concepts or designers of new software concepts for incorporation into the PRESS [31].

To gain a better understanding of the problems encountered in attempting to describe existing software concepts, a concerted effort was made to populate the PRESS with more concepts from the Steel Production domain. In total, fourteen CDFs were produced in the course of these initial studies. In addition, a large number of plain text documents from the Steel Production domain were installed in the PRESS for analysis.

Early on in the Practitioner project, software concepts from the domain of Steel Production were described using the original questionnaire. These consisted of fifteen questionnaires describing Order Processing in a Strip Processing Line. These provided initial test data for the project together with the partial thesaurus (of less than 1000 terms) covering the Steel Production domain developed at ABB [90]. In the work reported here, further concepts from the Steel Production were studied in order to augment the initial test data and to enable continued usage of the exist-

ing thesaurus in searching through the new concept descriptions. The primary goal though has been to gain experience in applying the the CDF in this domain. The case studies described here draw on material from three main sources:

1. Purdue Laboratory for Applied Industrial Control (PLAIC) steel plant control system description, the Williams report [169];
2. a complete set of project documentation for an annealing line plant control system, developed by ABB;
3. completed offers prepared by engineers in ABB's metallurgy division.

The Williams report is freely available as a published report; access to the other two sources has been freely granted by ABB with the proviso that the commercial confidentiality of the material is respected. (For this reason, the original material as recorded here has been edited. This has not restricted the research reported here in any way.) These sources were chosen for the case studies reported here for a variety of reasons. The Williams report has provided an opportunity to use the CDF in the description of generic software concepts. This report describes the tasks and functional specifications to be found in the general case of any steel plant control system. The more specific material from ABB was studied because of the project's goal of demonstrating the use of the PRESS to support offer preparation in the domain of Steel Production, and in addition, it provided descriptions of specific software concepts which could be related to the more general concepts found in the Williams report.

The Williams Report

The CDF was used to describe the software concepts in a Steel Plant Control System [169]. The system description developed at Purdue consists of several chapters; these are listed below and where the corresponding software concept has been described by the author using the CDF, this is denoted by the suffix of (C):

Chapter 1 - Some General Considerations Regarding Hierarchical Control
Chapter 2 - Summary and Statement of Purpose of this Report
Chapter 3 - Overall Management Information and Production Scheduling (C)
Chapter 4 - Area Control (C)
Chapter 5 - General Topics - Lower Level Members (C)
Chapter 6 - Coke Oven Unit Control and Supervision (C)
Chapter 7 - Sinter Plant Unit Control and Supervision (C)
Chapter 8 - Blast Furnace Unit Control and Supervision (C)
Chapter 9 - Steel Making Unit Control and Supervision (C)
Chapter 10 - Continuous Casting Control and Supervision (C)
Chapter 11 - Slabbing Unit Control and Supervision (C)
Chapter 12 - Hot Mill Unit Control and Supervision (C)
Chapter 13 - Pickling and Cold Mill Control and Supervision (C)
Chapter 14 - Finishing and Warehousing Control and Supervision (C)
Chapter 15 - Gaseous and Liquid Fuel Collection and Distribution Unit Control and Supervision
Chapter 16 - Steam and Electric Plant and Steam Distribution Control and Supervision
Chapter 17 - Electric Power Distribution Control and Supervision
Chapter 18 - A Description of the Computer System Contemplated for a Steel Mill Hierarchical Control System
Chapter 19 - Honeywell Plant Information Network as Prepared for the PLAIC Steel Complex Control Project
Chapter 20 - A Proposed Hardware and Software Description of the Steel Industry Hierarchical System Developed by The International Business Machines Corporation
Chapter 21 - Description of Example Steel Mill

Note that Chapters 15 to 17 address energy considerations that were not of immediate interest to the project's internal customer, ABB; so these were not converted to the form required by the CDF. The final chapters 18 through 21 are examples of typical systems including proprietary systems from Honeywell and IBM. They were not converted into CDFs owing to lack of time and effort and because of their specific nature.

The descriptions found in the Williams report are already in a structured form. Each subsystem is decomposed into its modular tasks at the supervisory level and its modular tasks at the control level, sometimes a further decomposition into direct control tasks is given. Where the subsystem consists of more than one supervisory or control unit, then the decomposition is given for each. Below are examples of decomposition found in the report. Mapping these onto the component concepts

of the CDF has not been problematic although it was possible when in doubt to consult experts in the domain for confirmation of the approach taken. For the overall acceptance of such work, the involvement of domain experts is important to ensure that obvious points are not overlooked.

The Williams report also describes the inputs, outputs and other data such as on-line variables used at both the supervisory level and control level for each subsystem. When treating each subsystem as a software concept, these various types of data can be mapped to the two categories of interfaces relatively straightforwardly as explained in the worked example below; however, the chapters give very little insight into whether or not the interfaces are external or internal in the context of the modular task decomposition. The assumption has been that they are all external.

A Worked Example of Conversion into the CDF

Here one chapter from the Williams report is considered in detail; its component parts are identified, and a rationale is given for mapping these into the CDF. The chapter to be converted here as an example describes *Hot Mill Unit Control and Supervision* (Chapter 12, Williams). The following extract taken from text of Chapter 12 describes the various tables included in the chapter:

> *Table organization is exactly as with previously considered mill production units. Tables 12-I and 12-II treat the supervisory level work. Tables 12-III to 12-VII handle the slab inventory functions; Tables 12-VIII to 12-XII are for the Reheat Furnaces; 12-XIII to 12-XVII for Roughing Stands; 12-XVIII to 12-XXII for Finishing Stands; 12-XXIII to 12-XXVII for Runout Table and Down Coilers and 12-XXVIII to 12-XXXII for the Coil Inventory Input.*

This description forms the basis for the top-level decomposition of the concept and the various tables were used to obtain a more detailed decomposition of the concept.

The list below gives the exact table headings as found in the text:

TABLE 12-I MODULAR TASKS OF THE SUPERVISORY LEVEL COMPUTER - HOT BANDS

TABLE 12-II DATA TRANSFERRED FROM OTHER AREAS - HOT BANDS

TABLE 12-III MODULAR TASKS OF THE CONTROL LEVEL COMPUTER SLAB INVENTORY

TABLE 12-IV ON-LINE VARIABLES TO BE CONSIDERED - SLAB INVENTORY CONTROL LEVEL COMPUTER

TABLE 12-VI INPUT DATA FROM OTHER SOURCES SLAB INVENTORY CONTROL LEVEL COMPUTER

TABLE 12-VII CONTROL OUTPUTS - SLAB INVENTORY CONTROL LEVEL COMPUTER

TABLE 12-VIII MODULAR TASKS OF THE CONTROL LEVEL COMPUTER REHEAT FURNACES

TABLE 12-IX DIRECT CONTROL TASKS - REHEAT FURNACE CONTROL LEVEL COMPUTER

TABLE 12-X ON-LINE VARIABLES TO BE CONSIDERED - REHEAT FURNACE CONTROL LEVEL COMPUTER

TABLE 12-XI INPUT DATA - OTHER SOURCES REHEAT FURNACE CONTROL LEVEL COMPUTER

TABLE 12-XII CONTROL OUTPUTS - REHEAT FURNACE CONTROL LEVEL COMPUTER

TABLE 12-XIII MODULAR TASKS OF THE CONTROL LEVEL COMPUTER ROUGHING STANDS

TABLE 12-XIV DIRECT CONTROL TASKS - HOT ROLLING MILL ROUGHING STANDS CONTROL LEVEL COMPUTER

TABLE 12-XV ON-LINE VARIABLES TO BE CONSIDERED HOT ROLLING MILL ROUGHING STANDS CONTROL LEVEL COMPUTER

TABLE 12-XVI INPUT DATA - OTHER SOURCES HOT MILL ROUGHING STANDS CONTROL LEVEL COMPUTER

TABLE 12-XVII CONTROL OUTPUTS - HOT MILL ROUGHING STANDS CONTROL LEVEL COMPUTER

TABLE 12-XVIII MODULAR TASKS OF THE CONTROL LEVEL COMPUTER FINISHING STANDS

TABLE 12-XIX DIRECT CONTROL TASKS - HOT ROLLING MILL FINISHING STANDS CONTROL LEVEL COMPUTER

TABLE 12-XX ON-LINE VARIABLES TO BE CONSIDERED HOT ROLLING MILL FINISHING STANDS CONTROL LEVEL COMPUTER

TABLE 12-XXI INPUT DATA - OTHER SOURCES HOT MILL FINISHING STAND CONTROL LEVEL COMPUTER

TABLE 12-XXII CONTROL OUTPUT - HOT MILL FINISHING STANDS CONTROL LEVEL COMPUTER

TABLE 12-XXIII MODULAR TASKS OF THE CONTROL LEVEL COMPUTER RUN-OUT TABLE AND DOWN COILER

TABLE 12-XXIV DIRECT CONTROL TASKS - HOT ROLLING MILL RUN-OUT TABLE AND DOWN COILER CONTROL LEVEL COMPUTER

TABLE 12-XXV ON-LINE VARIABLES TO BE CONSIDERED HOT ROLLING MILL RUN OUT TABLE AND DOWN COILER CONTROL LEVEL COMPUTER

TABLE 12-XXVI INPUT DATA - OTHER SOURCES HOT MILL RUN-OUT TABLE AND DOWN COILER CONTROL LEVEL COMPUTER

TABLE 12- XXVII CONTROL OUTPUTS - HOT MILL RUN-OUT TABLE AND DOWN COILER CONTROL LEVEL COMPUTER

TABLE 12-XXVIII MODULAR TASKS OF THE CONTROL LEVEL COMPUTER HOT BAND INVENTORY INPUT

TABLE 12-XXIX DIRECT CONTROL TASKS - HOT BAND INVENTORY CONTROL LEVEL COMPUTER

TABLE 12-XXX ON-LINE VARIABLES TO BE CONSIDERED HOT BAND INVENTORY CONTROL LEVEL COMPUTER

TABLE 12-XXXI INPUT DATA - OTHER SOURCES HOT BAND INVENTORY CONTROL LEVEL

TABLE 12-XXXII CONTROL OUTPUTS - HOT BAND INVENTORY CONTROL LEVEL COMPUTER

In the disposition of these tables, several conventions were employed. All tables describing data, such as "input data", "control outputs", "on-line variables" (Table 12-XXV), were taken as interfaces. The CDF requires that interfaces are identified as either *provided* or *required*; the convention employed was to treat all outputs as "provided" and all inputs as "required", other data such as on-line variables and data transferred from other areas were also treated as "required". As the various data tables distinguish the area of the application to which they are relevant, e.g. in the above list, Table 12-XXXII (reproduced below) describes control outputs for Hot Band Inventory Control, this was carried over into the interface name.

TABLE 12-XXXII

CONTROL OUTPUTS - HOT BAND INVENTORY
CONTROL LEVEL COMPUTER

1. Crane Movement
2. Conveyor Movement

So in the case of Chapter 12, Table 12-XXXII reproduced above gives rise to the following interface names:

1. Crane Movement (Hot Band Inventory Control),
2. Conveyor Movement (Hot Band Inventory Control).

In some cases, the Williams data tables have been quantified or parameterised. For example, in Chapter 12, Table 12-IV gives numbers following each of the items to indicate the typical number, as the excerpt below illustrates:

...

3. Slab Quality Input - Manual (1)
4. Slab Weight (1)
5. Slab Dimensions (1)
6. Crane Position (1/crane)
7. Crane Condition (1/crane)
8. Crane Speed (1/crane)

In other chapters, the parameterisation is used; for example, in Chapter 8 (Blast Furnace Unit Control and Supervision), Table 8-IX contains entries for:

33. Temperature of Furnace Coolant Exit of Each Section (m)
...
40. Identification of Torpedo and Ladle Cars (n x (1))
...
43. Tuyere Coolant Outlet Temperature (p x (2))

These are numbers required for each furnace.
Total will be double the given number.

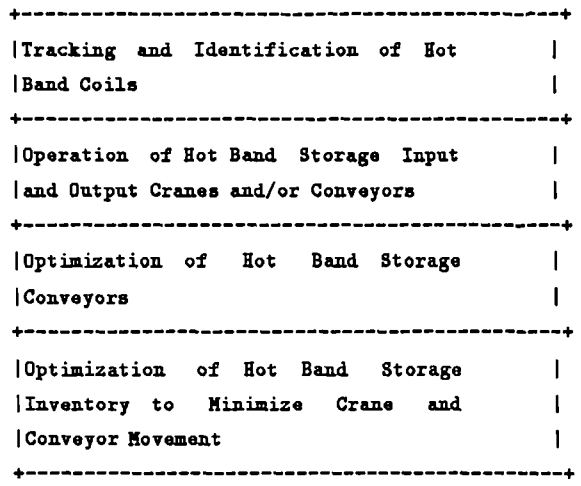
m is the number of coolant sections per individual furnace
n is the number of ladle and torpedo cars
p is the number of tuyeres

In converting the Williams text into CDFs, no use was made of these finer details although obviously such finer detail could form the basis of a more detailed description.

In order to distinguish the component parts of the concepts, the various tables describing supervisory and control tasks were used. Like the data tables, these tables include the area of application; so the convention employed was to take each table entry and append to it the area of application retaining the distinction between control and supervisory tasks and use this as a component descriptor. An example of such a table is reproduced here to illustrate the typical control tasks. This tasks come under the concept of Coil Inventory Input Control and can be found in its decomposition.

TABLE 12-XXVIII

MODULAR TASKS OF THE CONTROL LEVEL COMPUTER
HOT BAND INVENTORY INPUT



So in the case of Chapter 12, Table 12-XXVIII listed above gives rise to the following component parts assigned to the concept of Coil Inventory Input Control:

1. Tracking and Identification of Hot Band Coils (Control Task - Hot Band Inventory)
2. Operation of Hot Band Storage Input and Output Cranes and/or Conveyors (Control Task - Hot Band Inventory)
3. Optimization of Hot Band Storage Conveyors (Control Task - Hot Band Inventory)
4. Optimization of Hot Band Storage Inventory to Minimize Crane and Conveyor Movement (Control Task - Hot Band Inventory).

The chapters in the Williams report include a further decomposition of the control level tasks into direct control tasks. This could be used in CDFs describing the control level tasks as the basis for the component concepts description. In the work that was done with the Williams material, this depth of description was not attempted.

Twelve CDFs were completed using the Williams report. Figure 5.5 gives an overview of those resulting from Chapters 5 and 12. The CDF from Chapter 5 of Williams describes concepts found in any lower level control systems of which Hot Mill Unit Control and Supervision, the topic of Chapter 12, is a special case. Note that the decompositions are not exactly the same. In the CDF from Chapter 12, while there is one component concept concerned with lower level supervisory tasks, there are six component concepts concerned with lower level control tasks, one for each of the subareas of control of the hot mill unit. In the more general model recorded in the CDF from Chapter 5, there are only two component concepts: lower level supervisory tasks and lower level control tasks. The CDFs resulting from Chapter 5 were key to this study as they provide a general system model for the concepts covered in chapters 6 to 14 of the report, all of which are concerned with the control and supervision of various processing units of the steel mill.

The Continuous Annealing Plant Line - PODAS

In this case study, it has been possible to work from the complete set of documentation for one application from ABB; this documentation includes the original customer requirements (not in machine readable form) as well as all the ABB produced material. This material describes an annealing line control system, that ABB developed and includes the requirements specification, design and code documents (all available in machine readable form). The full title of the application is: Continuous Annealing Line Plant (ALP) - Process Operating and Data Acquisition System (PODAS). Annealing is form of Finishing carried out during Cold Working. From the standpoint of applying the CDF, the ALP material is ideally structured. It has two main subsystems, ALP_1 and ALP_3 (the latter being highly confidential is not available; the former is a merge of two earlier systems, ALP_1 and ALP_2). Within ALP_1, subsubsystems are denoted as ALP_1A, ALP_1B, etc and subsubsubsystems are denoted as ALP_1AA, ALP_1AB and so on. The Interfaces are distinguished as either Imported (which maps to Required) or Exported (which maps to Provided).

An outline of the main parts of the ALP_1 subsystem can be found in Figure 5.6 -

Figure 5.5: CDFs Resulting from Williams Report

Decomposition of ALP_1 - Continuous Annealing Line Plant PODAS. This decomposition follows from the overall hierarchical model described below although it is only concerned with control at the lower levels.

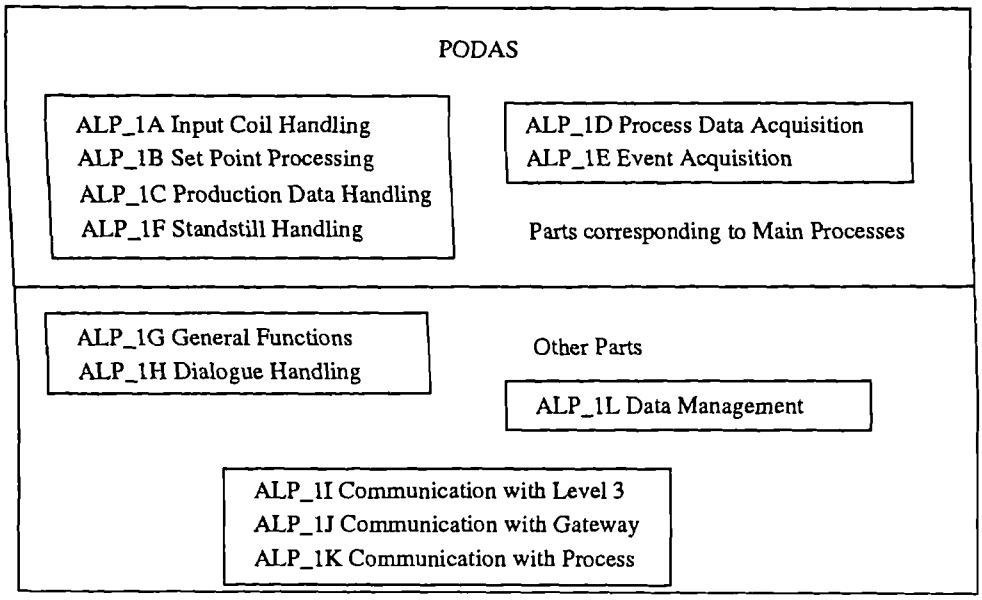


Figure 5.6: Decomposition of ALP_1 - Continuous Annealing Line Plant PODAS

A Worked Example of Conversion into the CDF

The text describing the overall PODAS system design specification was chosen for conversion into the CDF. As this text contains references to the major subsystems, it allows a link between between the overall concept description held in a CDF and subsystem descriptions held as files in the plain text document store with file names corresponding to subsubsystem identifiers. Each subsubsystem of ALP_1, that is ALP_1A through to ALP_1L, was taken as a component concept of the concept.

Although this text contained a straightforward decomposition of the concept into parts, it did not contain sufficient information about the interfaces for these to be described. Further information was obtained by examining the texts describing PODAS requirements in conjunction with the data flow diagram given for ALP_1. Each subsubsystem contains a description of its own interfaces distinguished as either imported or exported; however, from these descriptions, it was not always clear whether or not the interfaces of the subsubsystems were internal or external to the PODAS subsystem. Here greater domain expertise is required to identify these

PODAS interface bindings.

Four major interfaces identified were as follows:

- LASSO (Logging System for Investigation and Analysis) Interface
- Production Control System (Level 3) Interface
- Operator Interface
- Direct Control Level Interface

On reflection having filled in the CDF, these interfaces appear to correspond to the following other parts in the decomposition given above:

- ALP_1H Dialogue Handling,
- ALP_1I Communication with Level 3,
- ALP_1J Communication with Gateway, and
- ALP_1K Communication with Process.

More analysis is required to establish the correspondence between the interface concepts and component concepts which appear to provide these interfaces. This would have required CDFs for the component concepts to be developed so that their interfaces were identified and could then be linked to the main concept's interfaces. Analysis at this depth was not possible due to time constraints.

Offer Texts from ABB's Metallurgy Division

The source material which formed the basis of this case study has been various offer material in English and German prepared by engineers in the metallurgy division of ABB. Those offers studied in connection with CDF filling were as follows:

1. Annealing and Pickling Line Offer,

2. Electrolytic Tinning Line Offer, and

3. Hot Dip Galvanizing Line Offer.

A Worked Example of Conversion into CDF

The offer text for a Hot Dip Galvanizing Line Control System was chosen for conversion to CDF. This text is interesting in itself as it shows that reuse is already practised in the preparation of offers by ABB as the passage from this offer quoted below indicates:

> *The application software functions have been adopted from the functions realized within the* **Customer X** *project* **Project Y.**

Note that Customer X and Project Y have been substituted in this quote to protect ABB's customer's identity. The decomposition of the concept as recorded in the CDF is given by Figure 5.7 - Decomposition of Hot Dip Galvanizing Line Process Control. Although the offer text includes further decompositions of the concept's component concepts, these were not recorded in additional CDFs.

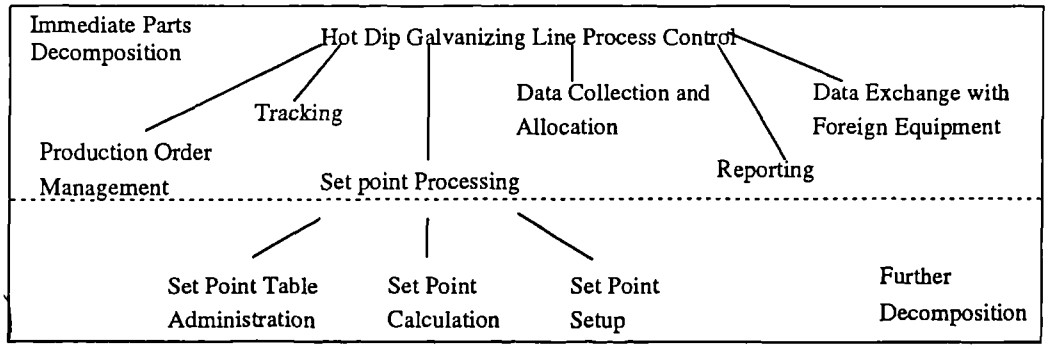


Figure 5.7: Decomposition of Hot Dip Galvanizing Line Process Control

As with the Continuous Annealing Line Plant, there was some difficulty identifying the concept interfaces, and here unlike in the case of PODAS, the difficulty was lack of detail (as one would expect in an offer to supply a system). The system proposed

in the offer text is restricted to Level 4 (i.e. Process Control); and in this case, Level 4a is intended as described below in the Hierarchical Model given in the Table 5.1. Here the level numbering is that used internally by ABB engineers; it is a reverse ordering to that found in the Williams report as the table shows. It is derived from a framework of levels developed by the steel producer, Hoogovens.

Experience Gained Using Known System Models to Assist in CDF Filling

As a result of this work, common system structures are a recognised starting point in the development of design frameworks of related software concepts; to quote Wirfs-Brock and Johnson:

> *Most people realize they need a framework when they notice similarities in existing applications.*

It is interesting to compare the decomposition given in the earlier questionnaire set filled out at ABB describing *Order Processing in a Strip Processing Line* [89] with those obtained in the later work described here. The earlier questionnaire set is limited to describing a strip processing line control system. Figure 5.8 - Decomposition of a steel mill control system - illustrates the decomposition graphically. The figure is a simplified version of the *Uppermost Level of General System Structure* taken from an earlier report [88].

The overall system decomposition can be compared to those given in the previous Figures 5.6 and 5.7 (PODAS and Hot Dip Galvanizing); there is a high degree of commonality in the essential elements:

1. automation of control;
2. data acquisition;
3. communication.

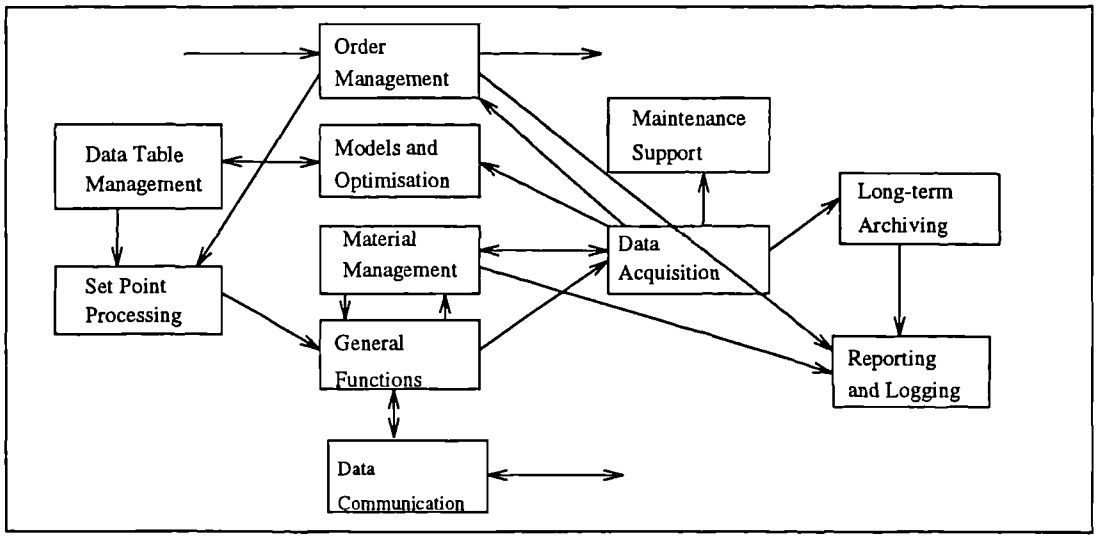


Figure 5.8: Decomposition of a steel mill control system

Notice all these figures have elements which are concerned with Set Point Processing i.e. with automation of control at the process level. All have elements concerned with data acquisition although it is variously termed as data collection, process data acquisition, etc. All have elements concerned with communication; these are most clearly identifiable in the PODAS figure.

The overall system decomposition is what one would expect given that the overall hierarchical model of control systems used are more or less equivalent as illustrated in Table 5.1. (Here Figure 1-14 from the Williams report has been adapted to include the levels used by ABB originally discussed in C2.1 [88]. These levels are also described in the text of the Hot Dip Galvanizing Line Offer prepared by ABB.) This model was taken as an initial design framework to guide the design-for-reuse work of identifying software concepts for reuse in this study of applying the CDF.

The availability of such models relieves the concept describer of much work. If the description of a new concept can be fitted to existing system design framework such as the above model of hierarchical control systems, then the task of describing the new concept can take place within established conventions such as recognised levels, known functions at each level, etc. Moreover, in order to understand the relationships between the parts of a concept, it is necessary to refer to the level of

Table 5.1: Models of Hierarchical Control Systems with Levels

Level ABB	Hoogovens	PLAIC	Focus of Control	Function	Applications
1	1	4	Corporation	Central Planning and Control	Long and Medium Term Production Planning Order Entry and Sales Order Dressing Weekly Loading Shipping Planning Invoicing
2	2	3	Product	Scheduling and Control	(Re)assignment of Material to Orders Mill Scheduling Quality Control
3	3	2	Production	Mill Control (Flow of material through the mill)	Data Distribution in the Mill Material Tracking Data Collection
4a	4	1	Plant	Process Control	Process Control (Grouped Setpoints) Data Logging
4b	4	1	Setpoints Group	Group Control	Setpoint Processing

control for which they are intended. Thus, design frameworks are useful to guide those designing for reuse as well as those designing with reuse.

However, there is no easy solution to the difficult problem of populating a Reuse Support System with reusable concepts in such a way that the concept descriptions facilitate reuse. The approach taken here has been to employ a canonical form for all descriptions of concepts - the CDF. Whilst this approach allows various descriptive methods and notations to be accommodated including natural language descriptions, it still imposes a constraint on the form of descriptions and this means that the existing software concept descriptions may require recasting into the CDF. Experience in the Steel Production domain has been that where systems have been designed in the framework of a common system model, the model itself provides insight into understanding the software concepts and their interfaces that are likely to found on examination of the existing software. This model constitutes a design framework and provides an important source of guidance for those attempting to develop software concept descriptions. It is especially helpful when attempting to identify the concept's interfaces; particularly where the description of the concept is restricted to a particular level or levels of the model.

One of the source concepts for design frameworks as used here has been the use of reference models in the development of communications standards. In Figure 5.9 -

OSI Layers Used as Framework for Relating Existing Network Designs (taken from [151]), the layers of the Open System Interconnection (OSI) model developed by the International Standards Organisation (ISO) are taken as a basis for comparing various network architectures. One view of this figure is that it constitutes a one-dimensional design framework for relating the concepts of ARPANET, SNA and DECNET.

<table>
<tr><th>Layer</th><th>ISO</th><th>ARPANET</th><th>SNA</th><th>DECNET</th></tr>
<tr><td>1</td><td>Application</td><td>User</td><td>End user</td><td rowspan="2">Application</td></tr>
<tr><td>2</td><td>Presentation</td><td>Telnet, FTP</td><td>NAU services</td></tr>
<tr><td>3</td><td>Session</td><td rowspan="2">(None)</td><td>Data flow control</td><td rowspan="2">(None)</td></tr>
<tr><td rowspan="2">4</td><td rowspan="2">Transport</td><td rowspan="2">Transmission control</td></tr>
<tr><td>Host-host</td><td rowspan="2">Network services</td></tr>
<tr><td rowspan="2">5</td><td rowspan="2">Network</td><td rowspan="2">Source to destination IMP</td><td rowspan="2">Path control</td></tr>
<tr><td rowspan="2">Transport</td></tr>
<tr><td rowspan="2">6</td><td rowspan="2">Data link</td><td rowspan="2">IMP-IMP</td></tr>
<tr><td>Data link control</td><td>Data link control</td></tr>
<tr><td>7</td><td>Physical</td><td>Physical</td><td>Physical</td><td>Physical</td></tr>
</table>

Figure 5.9: OSI Layers Used as Framework for Relating Existing Network Design

In the Steel Production domain, the levels of control provide the keystones of the framework relating various steel mill control systems. Unless a system is very simple, its description is bound to have levels of abstraction employed in describing it and the possibility of being characterised from more than one viewpoint. Domain analysis must focus on identifying relevant levels and/or viewpoints in a particular domain. Literature from the domain is probably the best starting point; and with the research described here, the work on exploration of terminology used in domain literature has been found to be a useful preliminary to such more focused study. The importance of consulting with domain experts cannot be overemphasized in this context.

Some considerations have been outlined here derived from specific case studied. The above discussion has been limited to brief experience of design-for-reuse in the domain of Steel Production where the interest has been to support the reuse of software concepts in the design of process control systems. Here the CDF fits

reasonably well with existing concept descriptions, but some problems noted above were encountered. The existence of an established model of Hierarchical Control Systems in the Steel Production domain went some way towards helping to solve these problems by providing the basis for a framework for conceptual understanding.

These initial studies have been based on an existing design framework. In the next sections, the results of further domain analysis in the domain of Steel Production are presented enabling further development of a design framework for software concepts in the Steel Production domain.

5.3.2 Domain Analysis Study Two - the Salzgitter Software Field Studies

This second study provided an opportunity to investigate software concepts as realised in the systems controlling an actual steel mill of a medium sized German steel producing company. One objective of this work was to study a system developed by a variety of suppliers and to determine whether or not the software concepts found would be recognisable within the design framework used in the initial studies. The part of the system studied was primarily developed by a major supplier of control systems in Germany which competes against ABB for business in this area.

Study Background - A Tandem Mill and its Associated Computer System

The cold rolling area at the Salzgitter steel mill is housed in a large hall. Much of the space is taken up with storage area for the input coils. The cold rolling line consists of five roll stands in tandem, i.e. in a line so that as the coil is unrolled and processed, it passes through the stands in serial order. In common parlance, such a cold rolling mill is described as a "tandem mill". The input coils are brought to the line by overhead lifting devices, and placed in the input area where each new input coil is joined to the previous by welding before being processed in the pickling

line; coils output from the pickling line form the input to the tandem mill. When each input coil is placed in the uncoiler before processing in the tandem mill, there is an opportunity for changing the set-up of the line if necessary. At each stand, there is a small display panel showing the current control values. The actual rolling line is concealed behind shutters, although it is possible for the operators to lift the shutters so that the strip being processed can be viewed.

The main control room for the line is located overhead above the processing line. Here the control operators have a number of video monitors recording what is actually happening on the processing line as well as a number of monitors displaying various screens of information about the state of the processing line. For example, data about each roll stand's current control values can be displayed. It is possible for the operator to select a particular screen display on an individual monitor. The displays are both analogue and digital in form with textual annotations; and good use of colour has been made in presenting the information.

The computer system controlling the line and supporting the control room displays is located in a more remote area. It contains the main system computers and system consoles as well as banks of microcomputer boards used in direct control. In this system, all the computer power is centrally located.

The area of cold working being controlled by the system consists of a pickling line with a tandem mill for cold rolling. This rolling line was first operational in 1963 and has always been computer controlled. The first enhancement to the computer control of the mill was in 1975; and the second and most recent improvements to the computer control were in 1985. These coincided with the installation of a fifth stand; the original line had four stands. With the installation of the fifth stand came the opportunity for new control systems at the lower levels to be installed.

The first step in the Salzgitter modernisation was the preparation of a call for tenders describing their requirements by the Salzgitter staff; in response, offers were received from three major vendors. The order for the new system went to the vendor who had provided the previous system because of their established experience and

knowledge of the existing system. However, the Salzgitter plant contains equipment and systems from all of these suppliers; and the management aims to maintain a balance amongst the major suppliers. Another major vendor has supplied systems in the roughing area, and a third has supplied furnaces as well as equipping the largest cold rolling mill and two smaller ones. Smaller companies have also supplied the plant; they tend to supply customised subsystems. The larger, more critical control systems must be reliably supported as the cost of the line breaking down and remaining idle is considerable. So for these more critical systems, the larger established vendors are preferred. For example, systems from major international suppliers are widely used by the plant for the higher levels of control.

The key elements of the control system taken from the call for tenders are as follows:

- Data Logger,
- Man-Machine Communication System,
- Optimisation Functions (controlling speed, size of slot, force, tension and thickness),
- Band Evenness Measurement and Control,
- Diagnostic System and Fault Reporting System,
- Microcomputer Level 2 Control System (covering Group Level Setpoint Control and Individual Level Setpoint Control).

The additional rolls added with the fifth stand in the 1980s have provided finer control. This stand has 6 rolls (i.e. is a 6-high stand) in contrast to the other stands which consist of 4 rolls each; in the stand, rolls are placed vertically one on top of another and depending on the number of rolls are termed "N-high". As explained in [74], tandem mill modernisation usually consists of replacement of the last stand by a 6-high stand; this enables operators to achieve a better set-up of the tandem mill by bending and/or shifting of the intermediate roll; this achieves improved flatness control. In addition with the new system, the thickness of the strips can

be monitored over time and when a break occurs, samples of the thickness are available for the past ten seconds either side of the break. This data can then be analysed to determine the cause of the break. Presently, in the rolling line, little information about the input coil is available other than its DIN (Deutsche Industrie-Norm i.e. German Industrial Norm) standard rating and perhaps a handwritten note to indicate if there might be problems. In future, a better integration of the hot rolling area and cold working is planned so that data about the coil from one area will be available during later processing. It should be noted that throughout the processing, coil data is collected for archiving for purposes of guaranteeing warranty and that such data must be kept for a seven year period.

The current computer system as installed consists of 50 Intel 286 based microcomputers for level 2 control and two mini computers for the associated modelling of the rolling process required for automation. As it stands, the rolling line cannot be operated in a manual mode; the microcomputers must be operational. When working, these process around 250 values per second. On the rolling line, the system uses sample data values to adjust its model of the process and then suggests new values for the control of each stand. The stand operator then has the option of accepting these or making adjustments to the values based on expert experience. These value changes, i.e. changes in the set-up, are possible with each new coil; and the average time between processing a coil on the line is between 3 to 7 minutes. Any adjustments made after the set-up are automatic and not subject to operator control.

In a more general discussion concerning the software, the system engineers pointed out shelves full of the system documentation as supplied by the vendor. This covers details of the Assembler, Fortran compiler, Linkers, etc. The application specific documentation consists of three binders; and this was of a high standard although the text and graphics were poor being based on "old line-printer standards". In comparison, the documentation of a subsystem supplied by a smaller firm was displayed, and it simply consisted of a listing with comments. Thus the planned development of CDF sets which would describe the main software concepts of the application and provide a high level conceptual view of the control system was regarded by the

Salzgitter staff as a potentially useful addition to the existing documentation. From a first inspection of this documentation, an overall system structure diagram was recorded. It is given in Figure 5.10 - System Structure Diagram for Tandem Mill Control System - with English annotations.

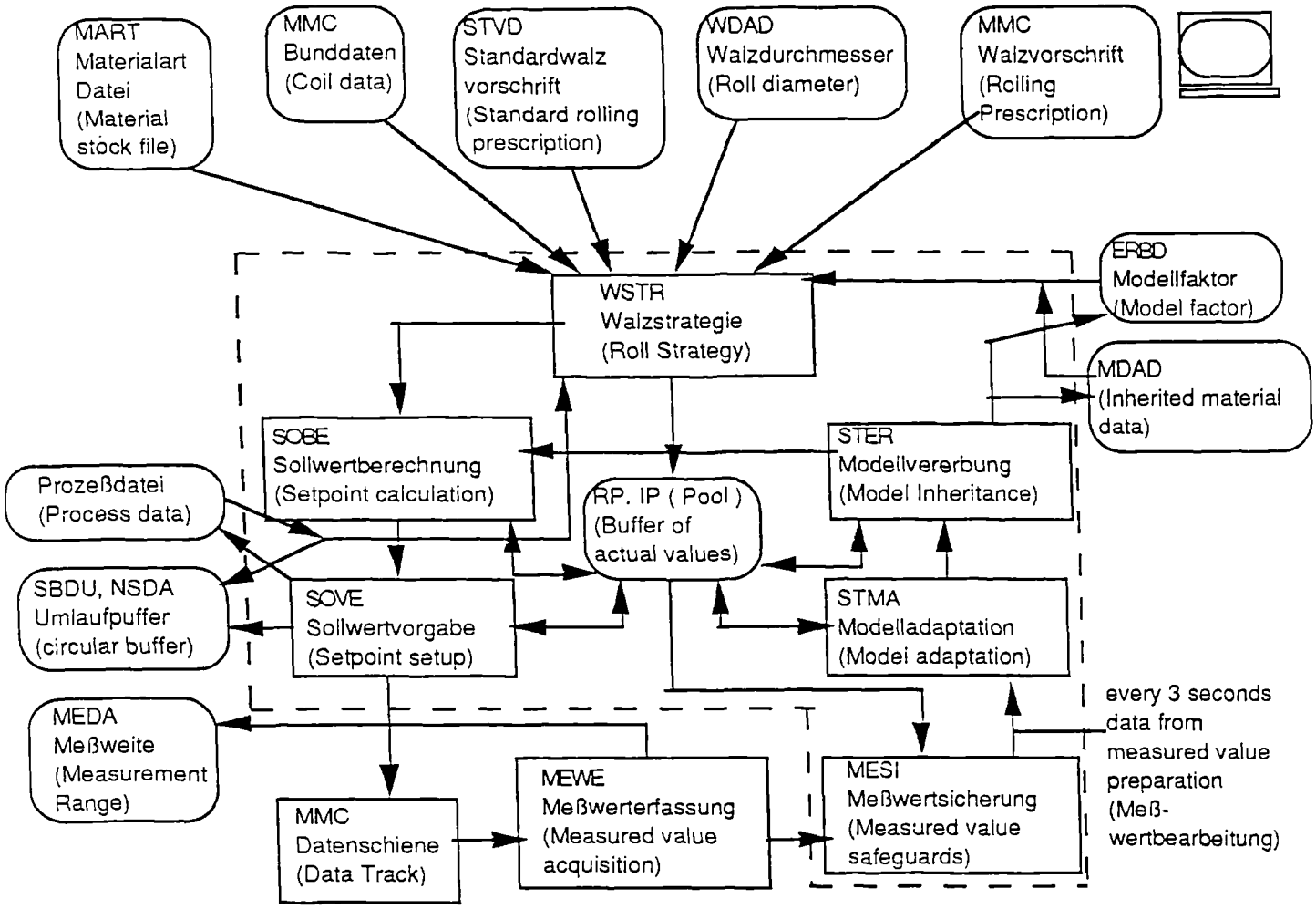


Figure 5.10: System Structure Diagram for Tandem Mill Control System

Details of the Software Concepts Studied

This study of software concepts in the area of Cold Working is complimentary to the one undertaken at ABB [89, 88]. The materials available from Salzgitter have been used to complete a set of CDFs describing a control system for a Tandem Mill. This has formed a basis for comparison with that already studied and described using the CDF, i.e. the concept of Pickling and Cold Mill Unit Control and Supervision based

on Chapter 13 of the Williams report. In the study, only a partial examination of the software system controlling the tandem mill in the area of Pickling and Cold Rolling (collectively part of Cold Working) has been made. More specifically, the focus has been on the software for determining the calculation of the draft rolling schedule (*die Stichplanberechnung* - abbreviated as SPB).

The study of the SPB has been made from three primary sources: the Peine-Salzgitter requirements document, the supplier's system specification and the supplier's documentation of the delivered system. In total, this study resulted in the filling in of 8 CDFs.

The Salzgitter concept set describes the following concepts:

C0 - SPB Draft Schedule Calculation, Version 1
C1 - WSTR Rolling Strategy, Version 1
C2 - SOBE Setpoint Calculation
C3 - SOVE Setpoint Setup
C4 - MESI Measured Value Safeguards
C5 - STMA Model Adaption
C6 - STER Model Inheritance
C7 - MMC Multi-Microcomputer Control, Version 1
C8 - MEWE Measured Value Acquisition, Version 1
C9 - SPB draft Schedule Calculation, Version 2
C10 - WSTR Rolling Strategy, Version 2
C11 - MMC Multi-Microcomputer Control, Version 2
C12 - PROST Process Control (Open Loop), Version 1

Note that those with version numbers have been installed as CDFs in the PRESS. In addition, a generic architecture for cold rolling mill control systems, more specifically tandem mill automation, given in [40] has been used to form the basis of following further concept descriptions:

C0 - Tandem Mill Automation Scheme, Version 1

C1 - Tandem Mill Schedule Adaption

The CDF recorded for the concept, Tandem Mill Automation Scheme, is given here in Appendix D.

The CDFs deriving from the Williams report are of a very general nature. In filling out the CDFs for the software concepts studied at Salzgitter, there was no straightforward link between the concepts other than the broadly similar structure of process control systems. The Bryant book which specially addresses tandem mill automation provides the intermediate level concepts to bridge this gap. This book had the express aim of developing concepts applicable to any system for control of a tandem mill, in particular, control via the on-line adaption of the rolling schedule. Based on an understanding built up over the course of these studies, Figure 5.11 - Conceptual Derivations for Concepts in Cold Working Control Systems - shows the links between these concepts in terms of their conceptual derivation (bear in mind that this does not of course represent the order of actually filling-in the corresponding CDFs).

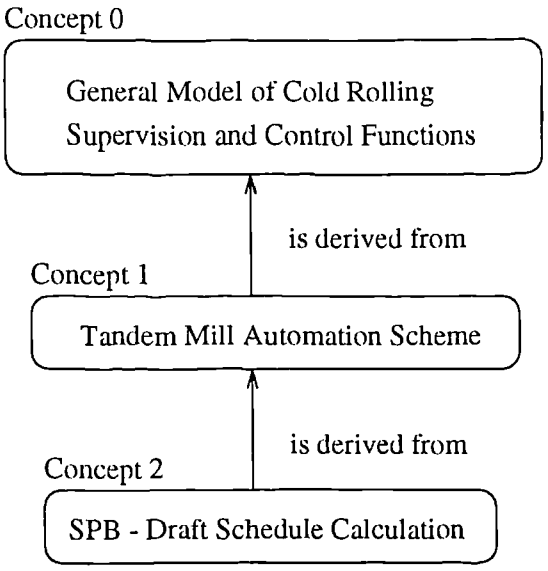


Figure 5.11: Conceptual Derivations for Concepts in Cold Working Control Systems

Also initially, the role of the Draft Schedule Calculation (SPB) software in the over-

all system was not so clear from the documents studied. In the supplier's system proposal, the SPB appears to communicate directly with the Multi-Microcomputer Control system (MMC); and the initial versions of the concepts reflect this. However, in the delivered system documentation, the SPB communicates with a system Process Control (Open Loop) (PROST). At first, it was thought that the name of the MMC had been changed to PROST, but further study showed that PROST was, in fact, an additional concept. An illustration of the connections between the SPB, Measured Value Acquisition (MEWE), the MMC and PROST is given in Figure 5.12 - SPB Located in the Automation System. Subsequently, new versions of SPB, Rolling Strategy (WSTR) and MMC have been made, and a description of PROST has been added.

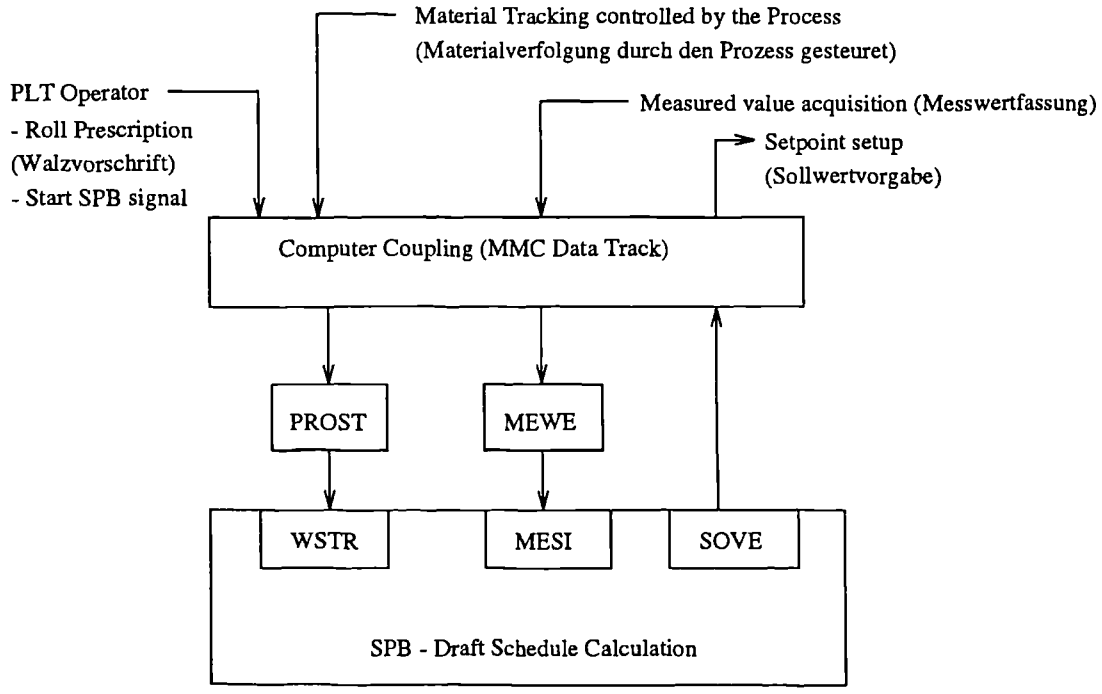


Figure 5.12: SPB Located in the Automation System

In the course of the study, a further report on control of rolling mills suggested that a different rolling system control concept derivation might also be relevant [117]. In this paper, a generalised system for automatic control of rolling mills is given. It

is based on a model which provided an abstraction across the areas of Hot Rolling and Cold Working, as well as different types of rolling process: reverse rolling and continuous rolling. This would result in the conceptual derivation found in Figure 5. 13 - Alternative Derivation for Rolling System Control Concepts.

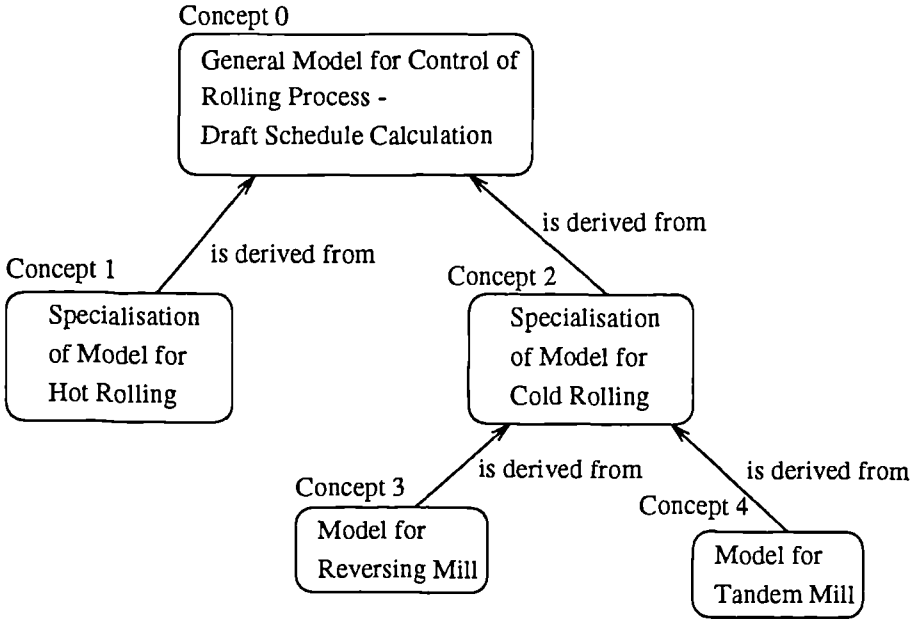


Figure 5.13: Alternative Derivation for Rolling System Control Concepts

This alternative model follows from a specialisation of the general case based on a substitution of the following concepts:

roll force and roll torque calculation,
temperature calculation, and
forward slip calculation.

Forward slip calculation for both hot and cold flat rolling process is based on Fink's formula. Rolling force and roll torque calculation for the hot-rolling process is based on experimental methods following Cook and McCrum's work. According to the paper cited, the improved Bland and Ford methods for rolling force and roll torque can be successfully used in the simulation of reverse cold rolling. The improved Bland and Ford method is the usual basis for simulating the roll force in continuous cold rolling; these equations of classical cold rolling theory were established in the 1940s with the work of Orowan.

This accords with the Bryant general model for cold-rolling-mill systems, and the suggested substitutions are based on common references, viz Bland and Ford *et al.* The main point here is not the details of theory, but to get across that a designer working in this area has established theory and concepts on which to draw provided the conceptual framework relating these is in place.

In the study of the Salzgitter SPB software, it was not possible to make a detailed determination of the algorithms employed for these concepts; however, study of the overall structure was found to correspond to that given in the Bryant concepts, and more generally to those given in the Williams report. From these considerations confirming the earlier domain analysis, it was concluded that the system concepts given in the Williams report would provide a reasonable base for further development of a design framework for this domain.

5.3.3 Final Study - Consolidation and Development of an Improved Framework

In this subsection, the relationships found amongst the concepts studied above and their CDF sets during their development and subsequent analysis are discussed, and a more detailed design framework for these concepts is developed. The aim of this study is to refine the conceptual foundation for process control applications in the domain of steel production discussed in the above studies. The remaining section will discuss how this framework might be used in the preparation of offers for new systems.

The most general architecture for control systems in Steel Production can be found in the Williams' report in Figures 5-1 through 5-5. These structure diagrams illustrate the typical modules found in any process control program in this domain and give data and control flows. These structures are described at Levels 1, 2, 3 and 4A as well as overall. The diagrams provide the basis for a very general system concept set comprising the following concepts:

C0 - Overall Process Control Programming System
C1 - Process Control Programming System, Level 1
C2 - Optimising Control Programming System, Level 2
C3 - Detailed Scheduling Programming System, Level 3
C4 - Overall Scheduling Programming System, Level 4A

This set forms the framework for the general concepts described in the CDF sets developed from the remainder of the Williams report and for the more specific concepts obtained from the domain studies of the ABB project documentation including offers and from the Salzgitter material supplemented by the Bryant material.

The framework presented here is based on synthesis of the PLAIC model of hierarchical control in a steel mill with refinements and elaborations deriving from theoretical considerations as well as the field studies of software used in practice.

The framework builds on the classic model of hierarchical control described in [100]. Mesarovic identified three essential characteristics of multilevel, hierarchical control structures:

- vertical decomposition;
- priority of action with higher levels controlling lower levels;
- performance interdependence (success depends on performance of the whole).

With respect to the application of hierarchies in process control, he distinguished three important types of hierarchies:

- levels of description or abstraction (strata),
- levels of decision complexity (layers),
- organisational levels (echelons).

An example of a type 1 hierarchical description is an industrial system modelled economically at one level, in terms of information processing and control at an intermediate level and in terms of physical processes at the lowest level. A self organising system can be described using levels of decision complexity; at the top level, the system is self-organising, but at the next layer, its elements may exhibit learning and adaption strategies which in turn are decomposed into selection strategies for control possibly based on some form of optimisation or regulation via direct control. The third type, organisational levels, is found in models consisting of a central element of control at the top with strictly defined chains of commands between the top and subsequent levels where elements at one level directly control those on the next level and reporting back from a lower level element to its immediate superior follows the same path. As Mesarovic points out, these three notions of hierarchy can be superimposed or combined in the same system description; and there are features common to all three types of hierarchies. Three features which he singles out are as follows:

- as one goes up higher in the hierarchy, the elements modelled become larger and the periods of time under consideration greater;
- in addition, at the higher levels, the pace of decision making is slower as the dynamics of change are slower and the exchange with the environment is less frequent;
- however, decision making at the higher levels is more complex as the descriptions are less well-structured and difficult to formalise due in part to uncertainties.

These features of hierarchical models make them especially suitable for application in the modelling of control systems in manufacturing. A four level model has been used extensively to provide standard terminology for describing the implementation of information and control systems in manufacturing [67]. Typically the four levels are distinguished as follows:

level A Regulatory and Sequential Process Control (i.e. control of field instruments and actuators in direct contact with the process)

level B Supervisory Control of level A (i.e. control of production through coordination of level A activities)

level C Plant-wide Information Systems (e.g. Product scheduling, Operations management, and Monitoring of Production)

level D Corporate Information Systems (e.g. Management Information Systems).

The specialisation of the hierarchical control model for the domain of steel production consists of four main levels: corporation, product, production and plant, with some separation of concerns within particular levels as suggested by the PLAIC model and the Hoogovens/ABB models already discussed in section 5.3.1. This dimension, levels of control, is considered together with a further dimension, areas of control, which provides a means of relating the system design to the physical areas of the steel mill as these map directly to areas of control for the software. These two dimensions present as separate aspects in the PLAIC model provide the basis of the design framework illustrated in Figure 5. 14 - Levels of Control and Areas of Control.

Each process area of the mill may be further decomposed into subprocess areas; for example, in the case of the Cold Working Area, the subprocess areas are as follows: Pickling line, Cold reduction mill, heat treating, Temper mill, Finishing, Product inventory and warehousing. Design in this dimension may be further supported by the existence of a relevant **reference-plant** concept. As explained in [91], a reference-plant concept provides a systematic approach to plant engineering consisting of a combination of recommended, pre-engineered plant arrangements (in building-block segments) to define the plant layout. It includes system descriptions to define all aspects of every plant subsystem in a standardised format, and standardised equipment specifications that are continually updated to reflect the latest industry standards and feedback from operational experience. Ideally an approved quality assurance program is also included. Although such models have been

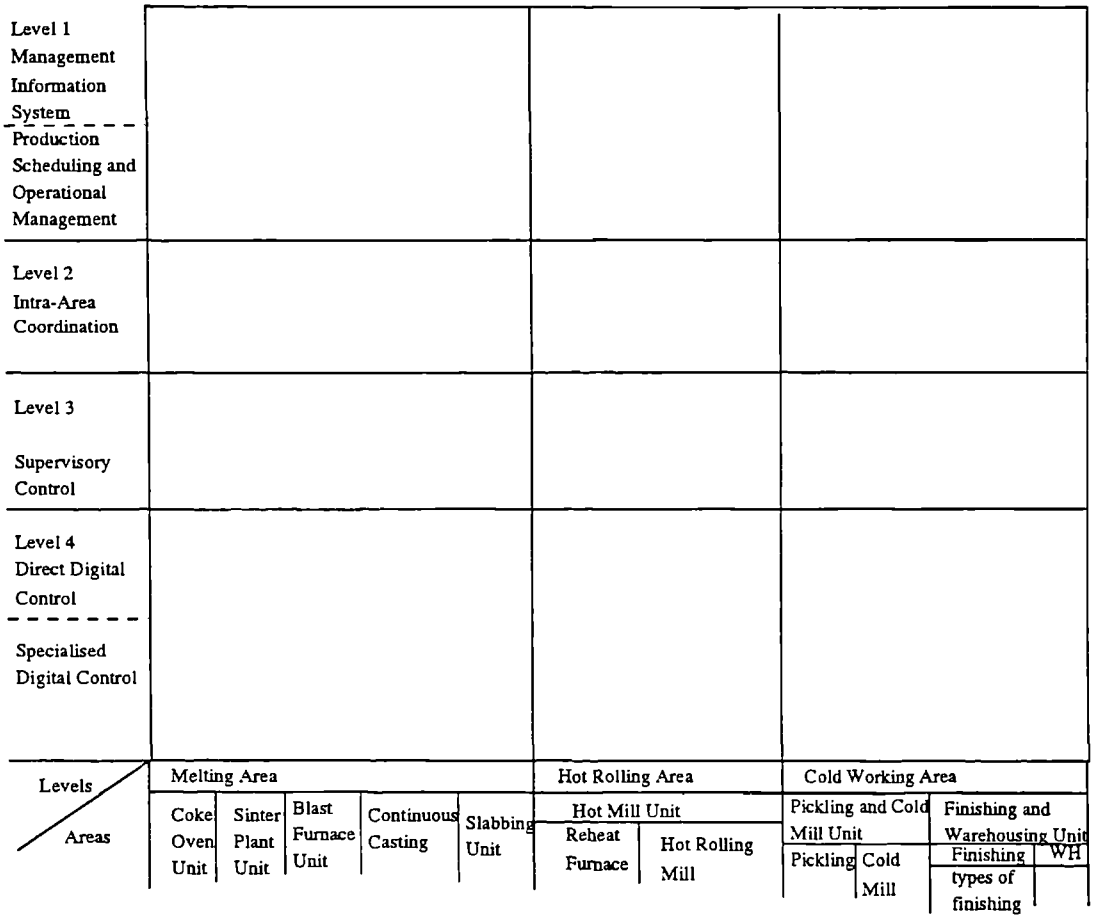


Figure 5.14: Levels of Control and Areas of Control

developed for power plants, it has not been possible to identify any published plant-reference models with the details outlined above for steel plants and none explicitly addressing the software requirements of a plant have been found. To better support reuse of designs, further domain analysis to fill in the framework along this dimension would be of value; however, such work is beyond the scope of this thesis.

In addition, three main flows may be considered for any system in the context of this framework:

- flow of information,
- flow of energy, and
- flow of materials.

These flows are identical with those found in von Bertalanffy's characterisation of systems given in his exposition of General System Theory [162] cited in [48], also see [163]. The identification of these flows is also recommended practice in engineering design as advocated by Pahl and Beitz [110]. Note that here this framework differs from the PLAIC model where energy is taken as a fourth area of control. However, conceptually PLAIC approach is unsatisfactory; as energy considerations are needed throughout the system. It is perhaps of note that this part of the PLAIC model was developed as an addition and that perhaps explains in part why energy considerations were not directly integrated into the main model. The approach taken here with three main flows characterising the system is confirmed by Heidepriem's view of a steel mill (shown here in Figure 5.15 - Steel Mill System's Characteristic Flows - based on a translation and simplification of one taken from his lecture notes by the author).

The basis of control is the flow of information in the system; and the objects of control, the processes, require energy and materials and may themselves produce energy for subsequent processes and result in outputs, e.g. products i.e. transformed materials. Given a particular system, identification of its flows of information, energy and materials, in the context of the two dimensional framework already presented

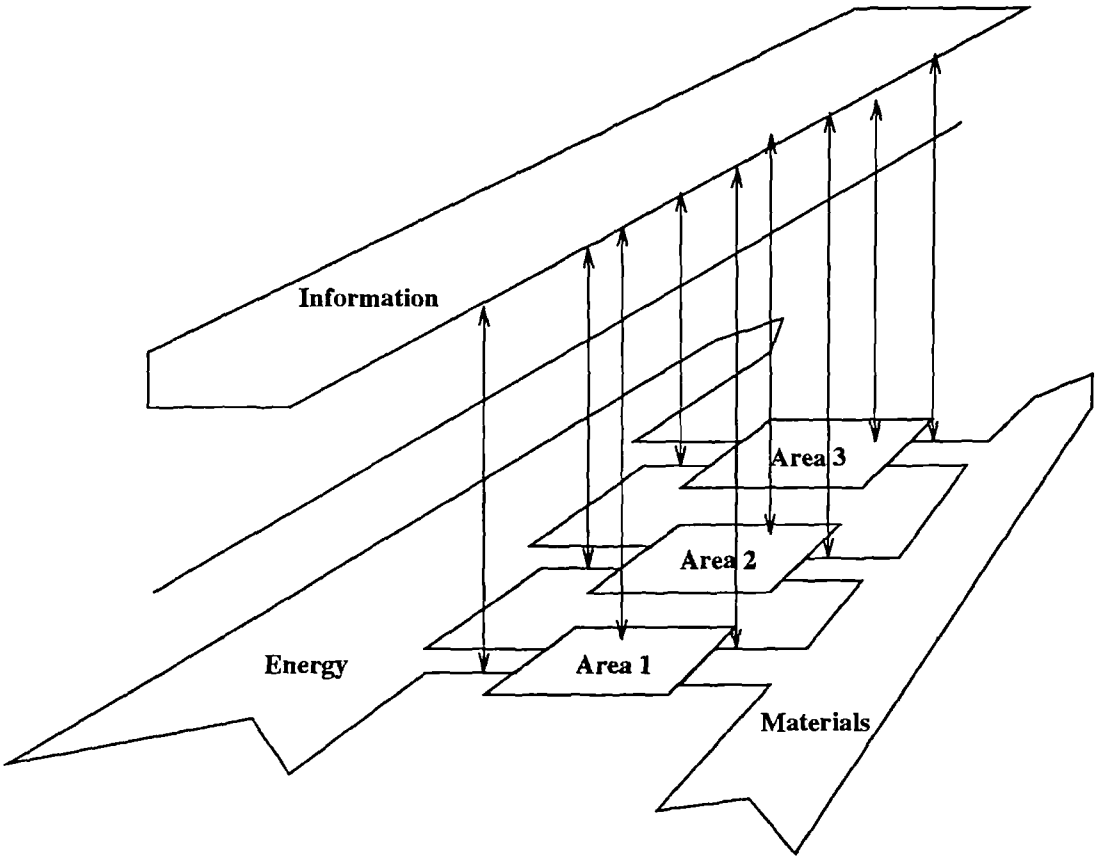


Figure 5.15: Steel Mill System's Characteristic Flows

enables a determination to be made of the degrees of integration amongst its parts in terms of these three flows. For example, in a system for controlling a cold rolling mill, information about orders may come from the top level, or it could be passed laterally from the hot rolling area control system. If a failure occurs in processing, the request for more materials could be handled locally or it may require passing a message to a higher level to initiate rescheduling of the order. If a system only addresses the lower levels of control, it may not be concerned with orders per se at all. For example, the determination of a draft rolling schedule is concerned simply with the calculation of control values to drive the actuators of a rolling line to meet specified requirements given in terms of physical characteristics of the rolled steel to be achieved; it is only if such a lower level system is seen in the context of a larger system description that the higher level concept of order can be understood as a determinant of the rolling process.

The hierarchical dimension, levels of control, coupled with the vertical dimension, areas of control, should not be seen as imposing a "straight jacket" of strictly hierarchical communication between levels and/or areas. The role of coordination within horizontal decompositions of the control hierarchy was already recognised by Mesarovic, and that this is now common practice is supported by evidence from Heidepreim's survey paper [74]. The framework presented here aims through the provision of a mechanism for describing the integrative flows of information, energy and materials to address this aspect of design. These flows are captured in the context of Practitioner concept descriptions through the specification of provided and required interfaces and associated interface bindings given in the CDFs.

A benefit of this framework is that it provides a basis for establishing the coverage of the software studies undertaken within the project. In Figure 5.16 - Populated Design Framework Relating Steel Domain Design Concepts, the framework is used for this purpose to relate the different CDF sets and earlier questionnaire sets developed. It should be noted that these studies have primarily addressed the heart of the AISE layered model of Process Control Software, i.e. the process technology software. However, within the framework given, hierarchies of domain data objects within the corresponding control levels and areas can be identified; similarly, levels

of Man-Machine Interaction concepts and Communications concepts could also be identified.

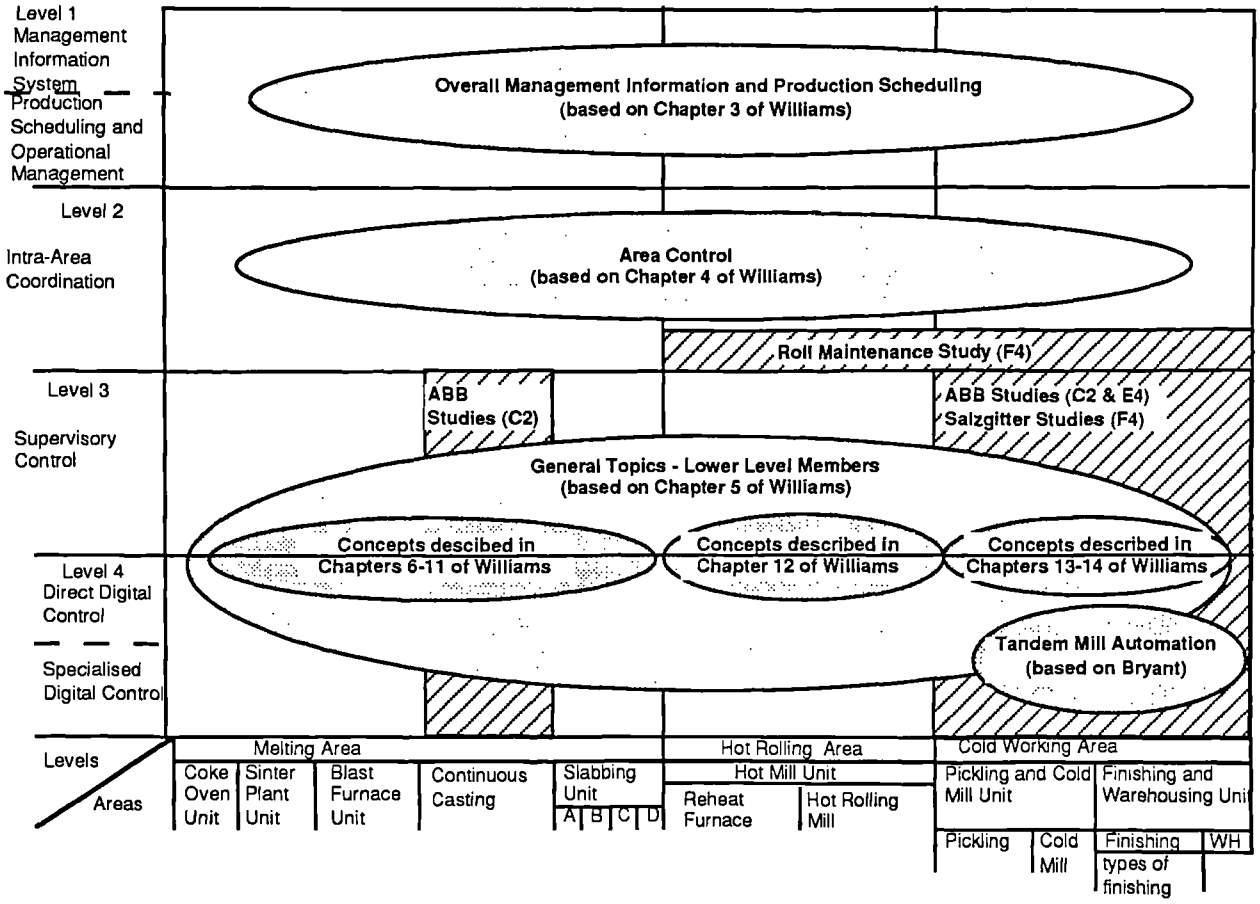


Figure 5.16: Populated Design Framework Relating Steel Domain Design Concepts

The chief benefits though come from the use of the framework as a record of conceptual understanding of designs found in the domain. As a result of the initial studies, it was argued that the existence of such models greatly assists in domain analysis; and in Chapter 6, the utility of such frameworks as carrier of high level design concepts providing the necessary levels of abstraction to support reuse of design concepts will be outlined.

A more general account of the benefits of hierarchical structures (of which the framework presented here is an example) in design is given in Mesarovic. He notes the role of such models in providing frameworks for system integration as well as other benefits such as stratification of control, limitations on building modules, better

utilization of resources and overall increased system flexibility and reliability.

Relation of framework to other developments

The specific framework developed here for the Steel Production domain can be related to more general research addressing architectures for Computer Aided Manufacturing in Europe and the USA [11, 44]. The ESPRIT Project AMICE is a collaboration amongst European industrial and academic partners; and this work on Open System Architecture for CIM has been fed into various national and international standardisation efforts. The AT&T research by Campbell *et al.* whilst the effort of an engineering research centre of a large multi-national company, nevertheless also explicitly addresses the need for and benefits of standardisation in this area. Of these two architectural frameworks, the AT&T architecture is closer to the framework developed here. The four levels of the PLAIC/Hoogovens model used in the framework can be seen as a specialisation of the more general seven level AT&T model. The AMICE OSA for CIM takes a more generative approach to the specification of system architecture; and its modelling of open systems starts with the enterprise and is at a much higher level of abstract than the process control systems considered here.

5.4 Consideration of Design-with-Reuse Using Steel Production Concepts

Throughout the domain studies described above, the CDFs developed were being installed in the PRESS and additional terminology was being added to the associated PRESS thesaurus. Although not made explicit in Chapter 3, the PRESS, the reuse support system developed as part of the Practitioner project, was developed to support the CDF rather than the original questionnaire. As a result of the PRESS development by other members of the Practitioner project, it was possible

to process the CDF in the same way as described in Appendix A with respect to the questionnaire when installing CDFs in the PRESS concept database. All the discussion in Appendix A with respect to the questionnaire applies equally to the CDF. In particular, the links between indexing terms in the thesaurus and the questionnaire were established between the thesaurus and the CDF so that CDFs installed in the PRESS could be retrieved using the thesaurus as explained in Appendix A.

The population of the PRESS formed the basis for various demonstrations. Demonstrating the payback on reuse in the application domain of process control in Steel Production was rather difficult as realistic software developments are likely to take place over a number of years and demonstrating design-with-reuse would have required a timescale greater than the remaining lifetime of the project. However, within ABB, the process of *offer preparation* was identified as one where reuse of software concepts could have an immediate impact. Responding to calls for tender, by preparing *offers*, i.e. proposals to build the required software, is an area where reuse typically has not been considered; and yet it provides a short enough illustration of reuse without being too simplistic for demonstration purposes while at the same time bearing enough resemblance to the design process as a whole to make realistic use of the CDFs resulting from the author's work. It was also possible to build on informal reuse of material in offer preparation already practiced by experienced engineers in the metallurgy division of ABB.

Below one of these demonstrations developed by the author in collaboration with a domain expert from ABB is described in more detail as it illustrates how design-with-reuse could be supported as a result of the author's work using the CDF to capture software concept descriptions from the domain of Steel Production.

More generally the use of models in control system design is also considered with reference to use of models in cold rolling mill control; and a role for the CDFs and associated framework is identified.

Table 5.2: Terms of Interest in Customer's Text

galvanized	galvanealed	coating
models	coils	strip
setpoint calculation	setpoint generation	data acquisition
time related	strip related	HDN (Hoogovens Data Network)
communications link	coil related data	order related data

5.4.1 Offer Preparation Using the CDFs

At the Sixth Practitioner Project Review, a demonstration was made by the author to show how the PRESS could be used to support the work process of Offer Preparation [29] using the CDFs from the first domain study reported here. The population of the PRESS with materials from the relevant area of Steel Production control systems, that of Cold Working, was first explored; and then an offer was partially prepared on the basis of a customer's requirements for a galvanisation control system (included here in Appendix E). This requirements text may be taken a typical example of a customer's requirements in this domain.

Informal Exploration of Concepts

The informal exploration of the concept base was made using terms of interest in the requirements that were highlighted by the project's domain expert, Johannes Keilmann of ABB's Metallurgy division. These terms are given in Table 5.2 and italicised in the text as it appears in the appendix.

By inspecting the thesaurus, concept base and text document store, it was possible to demonstrate that the PRESS contained potentially relevant material for this case. Figure 5.17 gives an overview of relevant control system concepts from the Williams report and illustrates the thesaurus relations that link galvanizing and annealing. They are both forms of *Finishing*; the relevant CDF is derived from Chapter 14 of the Williams report. One of the terms of interest in the text, *galvanealed*, although not present in the thesaurus could form a starting point for exploring the relevant thesaurus relations.

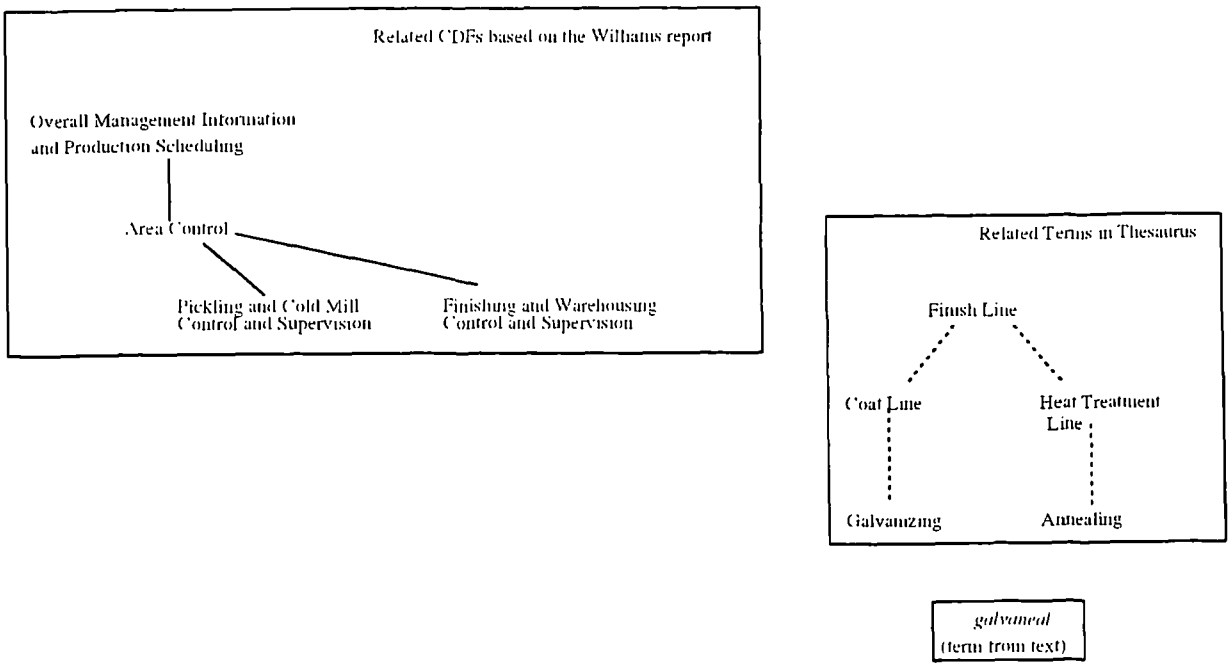


Figure 5.17: Related CDFs and Related Thesaurus Entries

Table 5.3: PRESS Contents Retrieved

CDF	Plain Text Document
ALP_1 Process Operating and Data Acquisition System (PODAS)	design text - alp1.dok
Hot Dip Galvanizing Line	offer text - a46.txt
Pickling and Cold Mill Unit Control and Supervision	Williams report - Chapter 13
Finishing and Warehousing Unit Control and Supervision	Williams report - Chapter 14

The PRESS contents retrievable using the terms "galvanizing" and "annealing" include several CDFs and plain text documents are listed in Table 5.3.

Inspecting the last two concepts in the context of the related CDFs from the Williams report allows the reuser to see where the proposed system is located. This system is concerned with control at the lower levels as can be seen from the fact that both concepts are derived from the concept, Lower Level Supervisory and Control Systems. This relationship is shown in Figure 5.17.

Offer Preparation Process Outlined

The work process of offer preparation as carried out at ABB's Metallurgy division is described in Figure 5.18 below.

There are two major activities within this process where the thesaurus and CDFs are used, first, during understanding and analyzing the customer's requirements, and second, during the initial formulation of the design of the proposed system. Here the demonstration was concerned with the use of the thesaurus and concept base to support the initial formulation of the design by showing how material describing concepts relevant to the proposed system could be found. Of course, the process of offer preparation is very dependent on the quality of the customer's expression of requirements found in the call for tenders. The expression of requirements can vary from one or two pages, as in this case, to one or more substantial documents.

In this demonstration of offer preparation, an analysis of the customer requirements was made using the clustering technique of Alexander described in [9]. This enabled three main requirements to be identified:

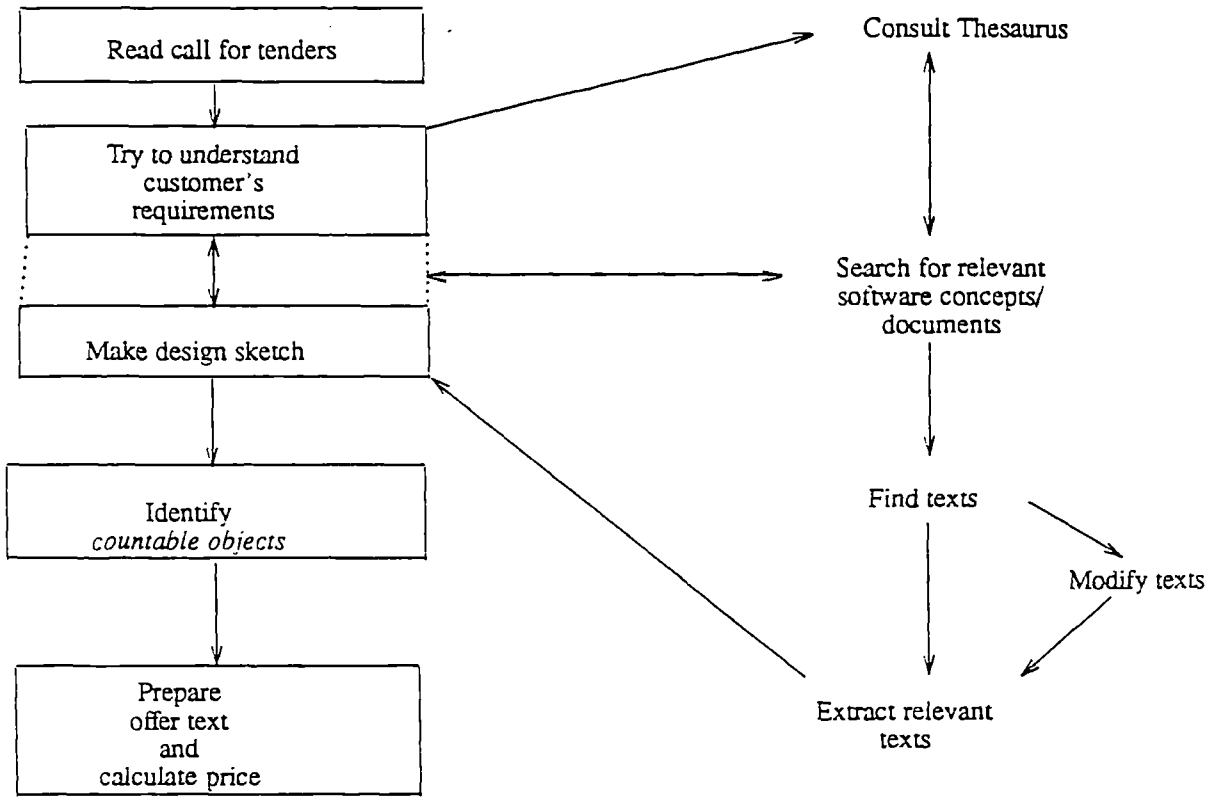


Figure 5.18: The Work Process of Offer Preparation

- Automation (Set Point Processing) (Requirement 2.1 in the text)
- Data Acquisition (Requirement 2.4)
- Communications (Requirements 2.2, 2.3, 2.5 and 2.6)

For example, in the text, there are several requirements which can be clustered together under the concept of communications: Man Machine Communication, Alarm System (selective presentation of alarms and messages), Interface with Production Control System and Input/Output System (including remote I/O). These three main requirements identified correspond closely to the three search phrases suggested by the domain expert from ABB which were as follows:

- set point calculation
- data acquisition
- communications link.

In the search for CDFs, the phrases of the domain expert were used. All of the search phrases were demonstrated to be adequate for identifying relevant material to be used in the preparation of an offer, i.e. they resulted in the retrieval of some of the same CDFs that were retrieved earlier in the exploration phase. The results of using these search phrases are discussed below in greater detail with respect to the first and last requirements.

Results of Set Point Calculation Search

The concept of setpoint or setup (both terms are used interchangely) is as follows:

> Input to the strip processing line takes the form of coils. These vary considerably (see Section 1. Basic Data and Requirements for DVL2 in Appendix E). For each new coil input, a different setup may be required

depending on the properties of the new coil. This setup may be generated using models and/or tables. (Tables were more common when computing power was limited; now more sophisticated setpoint calculation is possible using models.) For each input coil, the setup of the line can be changed with respect to various variables such as tension, furnace control, etc depending on the variation of the input coil's width, thickness, temperature, etc and the output coil required.

Setup data is required for the lower level control of the strip processing of each input coil to produce the output coil required. Here is it relevant to refer to Table 5.1 which shows the Purdue levels of hierarchical control and the corresponding Hoogovens levels.

It should be noted that *setup* and *setpoint* are used interchangely in the above description. The texts to be searched contains various forms of both terms, such as:

set up
set-up
setup
set point
set-point
setpoint.

From the entry in the PRESS Thesaurus of "set point", using the PRESS ThesaurusTool, it can be seen that there is link with the UF (i.e. use for) term "set up data".

Using the PRESS SearchTool, if one executes the following CCL:

```
find set#up or set#point or set point+UF
```

this will ensure that both cases are covered and a number of CDFs will be found; these will be from both the ABB material and the Williams report. The five CDFs retrieved were as follows:

- Coke Oven Unit Control and Supervision
- TAUFT
- SWDAT
- ALP_1 Process Operating and Data Acquisition System (PODAS)
- Hot Dip Galvanizing Line Process Control

The first three are not relevant as they relate to control of units in different areas which also happen to involve set-point calculations. For example, Coke Oven Unit Control and Supervision comes under the area of Melting, not Cold Working where Galvanizing occurs. The offer preparer was able to determine this by brief inspection of the contents. The earlier exploration leads the preparer to examine the CDF based on the offer for a Hot Dip Galvanizing Line, from which an outline of the set point processing can be extracted for incorporation into the offer.

Using the PRESS TextRetrieval tool on documents, the preparer was able to retrieve further material from the original ABB offer texts and chapters from the Williams report. This was done during the demonstration in order to show how links to other documents in the CDF can be followed up.

Results of Communications Link Search

This is probably the most standard part of the offer. Note that the offer is for a system that only operates at Level 3 (ABB/Hoogovens), i.e. Level 2 of the Williams model, and below. Effectively these are the levels where real-time considerations apply. The communication link is to the higher levels. Here the ALP PODAS concept was found in the concept base using the search term: communication. On inspection, CDF links to the relevant plaintext documents were noted and the TextRetrieval tool's file selection feature was used to select the relevant ALP project documents for reuse.

Building up an Offer from the Retrieved Material

As a result of the searches described above, relevant material was extracted and pasted into a window running an editor with a skeleton offer. The system requirements make no mention of any need for material tracking in the system to be developed, but inspection of the CDFs retrieved shows that a system component to handle material tracking is needed. This is an example of the sort of requirement often omitted by the customer which the system designer must supply. The author although not an expert in the design of steel control systems was able to reason that a material tracking subsystem was required by inspection of the decomposed concepts of similar CDFs and then confirm this by consulting a domain expert. As the expert remarked, an experienced engineer would realise this. In this demonstration, the author had no such experience, but was able to recognise the omission by referring to existing concepts descriptions of related systems. Further preparation of the offer text as outlined in the figure above was then able to take place, details of this are not relevant to the discussion here. The offer that was prepared in this demonstration would require very little editing to be itself be converted into an additional CDF.

5.4.2 Use of Models in Control System Design

Although the CDFs described here have not been employed in any realistic design, the use of models such as those described here has been a recognised design practice in this domain for some years. In the early seventies, the Industrial Automation Group at Imperial College working with GEC (then GEC-Elliot) and British Steel gave consideration to the use of models in the design of cold rolling mills control systems [57]. The design process as described by Edwards *et al.* consists of first establishing complex system models based on the plant design, design data and plant data. These are subsequently refined into simplified models whilst retaining the basic essentials but imposing "reasonable" limits. The simplified models are then subjected to detailed analysis, both static and dynamic checking, and finally

pilot trials. A key point from this description is to take note of the experimental nature of system design in this domain within the context of established system models. Such established models could be obtained from consulting a populated design framework such as the one developed here.

Figure 5.19 - Modelling in Control System Design - adapted from Bryant gives an overview of this approach to design and indicates where the CDFs describing typical systems models could be used.

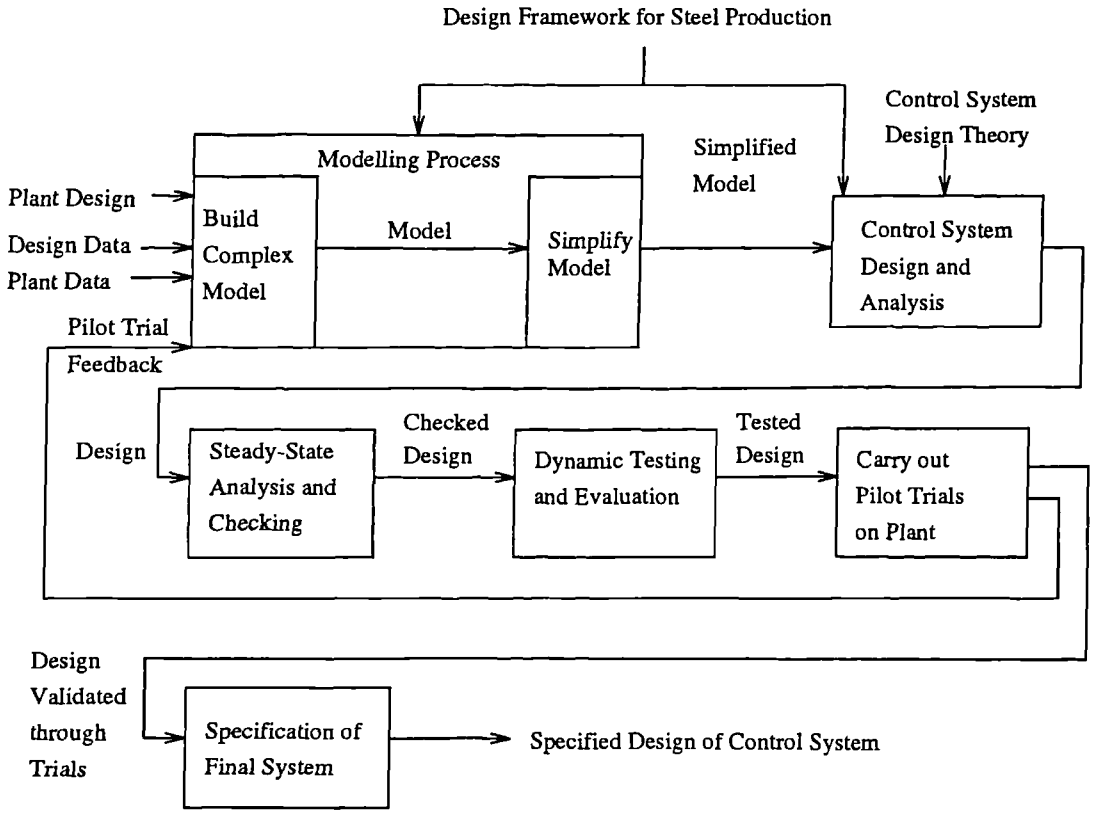


Figure 5.19: Modelling in Control System Design

This use of models was also seen in the Salzgitter study where the system being studied was subject to development by introducing a new rolling model into the software using this same approach of experimental trials.

5.5 Results of CDF Application in the Steel Domain

The author in the course of these studies consulted several volumes of domain literature and software documentation from the steel domain. While the number of software concepts identified has been quite large, CDFs have only been recorded for a small number (less than fifty). However, a design framework relating the results of these studies has been established and this work could be developed now by working from the established system models recorded with the CDF. Here only a small part of the work has been developed in field studies on tandem mill automation concepts as realised in a specific system.

In practice, there were problems found with the existing system documentation, but enough information was available to allow the concepts to be described using CDFs and related to more general concepts from the earlier domain study.

Many CDFs were only partially completed by the author in the course of this work. The effort to complete these CDFs would be considerable, and such work is considered beyond the scope of this thesis. Nevertheless, sufficient concepts have been described to characterise this domain and provide the basis for a demonstration of design with reuse, albeit the limited case of reusing concepts in offer preparation, described above.

The CDF remains a very minimal form and while this means that it is relatively easy to employ in structuring concepts descriptions, it also means that it results in no added formalisation of the concept descriptions which could support a more formal analysis of the underlying domain concepts realised in its software. This discussion will be continued in the following chapter.

The most difficult part of applying the CDF consists of establishing relations amongst various domain concepts, and here the maturity of the domain being studied is the key factor. Although standard textbooks were not available for the steel domain, its

design traditions have been given order by the availability of common models based on the principles of hierarchical control as reported in the final study.

It has been difficult to assess the success with which the CDFs resulting from this work can be used to support design with reuse in practice owing to the timescale of software projects in the steel domain. However, the author's work on concepts in tandem mill automation did result in bringing to light more general concepts which were useful to the software engineers at Salzgitter engaged in the process of updating the rolling models in their tandem mill control system. The acknowledged reference work for tandem mill automation is the book by Bryant *et al* (this was confirmed by the author consulting Professor Heidepriem, an expert in steel mill automation); but, prior to the author's work to establish the linking concepts between the PLAIC model of cold rolling control found in the Williams' report and that found in the existing system at Salzgitter, this book which describes the relevant concepts was not known to the Salzgitter staff. Without this work, it would not have been brought to the attention of the Salzgitter staff, and so subsequent benefits deriving from the design with reuse of these concepts would have gone unrealised.

Many application domains like steel employ software systems that are developed throughout their lifetime and are subject to development by their users. Explicitly recording the software concepts in these systems and relating them to more general concepts through the CDF has provided the conceptual basis for such evolutionary development based on concept reuse as the experience at Salzgitter shows in the specific case of tandem mill automation software.

5.6 Conclusions

In this chapter, the CDF has been applied in a series of domain studies and its usage to describe software concepts at various levels of abstraction has been investigated. The studies in the domain of Steel Production reported here have enabled the software concept database of the PRESS to be consolidated and expanded. This work

was undertaken with the goal of developing a better understanding of how to use the CDF in domain analysis of the software concepts as realised in existing software. It has been shown that the CDF can be used to describe generic system models prior to carrying out more detailed software concept description. The iterative nature of domain analysis has been recognised as a result of this work; and in practice the domain studied was found to be much more complex than that of compilers. However, it was possible to use the CDFs resulting from these studies as the basis for abstracting a design framework for relating typical system concepts found in designs from this domain. Further studies have analysed concepts found in a working steel mill and it was possible to relate these to the more general concepts studied earlier. A final study developed the foundations of the design framework. The resultant framework partially populated with CDFs is a major result of this research. This formed the basis for a demonstration where the work process of offer preparation was supported by the possibility of reusing material from CDFs resulting from this work. A more detailed evaluation of the work reported here will be made in the following chapter.

Chapter 6

An Evaluation of the Concept Description Form and its Application to Support Reuse

This chapter is concerned with evaluation of the work described in the last three chapters. These describe the development of the CDF and its application in a series of domain studies. The results of the domain studies in steel were employed in various demonstrations. One of these has been described in some detail in Chapter 5. Here a critical examination of the CDF development and associated application studies is made in order to determine the longer-term prospects for the CDF and wider application.

Experience in applying the CDF is discussed and related to general difficulties in domain analysis. The scope for replicating the application of the CDF to support domain studies in other application areas is given consideration. In addition, the problems of developing and maintaining a set of CDFs over time is briefly considered, as this was a primary concern of the industrial partner, ABB, with respect to the practicality of implementing reuse using a standard form. The minimal nature of the CDF is recognised and its potential development towards a more formal form

is given consideration. The advantages and disadvantages of moving towards more standardised forms of software descriptions are discussed.

The support that the CFD gives for recording designs structures is discussed given that the role of known structures has been recognised as important in design. Both Randell and Shaw have recognised this; and this insight can be found by re-examining the works of Alexander and Jones to which Randell refers. In using the CDF to record the results of domain analysis, a systematic approach to design-for-reuse has emerged based on the use of general system models. Through studying sets of CDFs, in the industrial domain studied, it was possible to develop an associated design framework to relate the design concepts of that particular domain. In Chapter 5, the role of such a framework in more detailed analysis of the domain was discussed and a demonstration was made using sets of CDFs to support reuse in offer preparation. Here a more general discussion of the utility of design frameworks based on sets of CDFs to support concept reuse can be found in conclusion of this chapter.

6.1 CDF Development

The development of the CDF arose out of the research with the Practitioner project on how best to describe software concepts as realised in existing software in order to support their reuse. Originally, a questionnaire was developed by the project to use in obtaining an abstracted form of the software documentation describing a particular software concept. The development of the CDF described here was as a result of problems encountered with the questionnaire. Based on an examination of interconnection languages and other developments to support reuse at a high level of abstraction, the author identified six major requirements that supporting conceptual reuse imposes on any language. The CDF was then developed from the earlier questionnaire to meet these requirements.

In Chapter 3, the various parts of the CDF were linked with the six requirements

identified; and the structure imposed on software concept descriptions using the CDF in a small scale application was illustrated in Chapter 4. The various relations amongst a set of software concepts that can be described using the CDF were outlined. In following section, experience gained in actually applying the CDF to record the descriptions of existing software concepts and relations amongst them will be addressed.

In the remainder of this section, the CDF development is considered independently of its application. Clearly one view of the CDF is that discussed in Chapter 3 and depicted in Figure 3.5 i.e. a means of recording abstracts of software life cycle documents. This enables the CDF to be used as a means of linking the concept description with the software life cycle documentation (Requirement R6). However, the other requirements which focussed the CDF development give other views of the CDF. In particular, these other views arise because references to other CDFs can be made in the description of a particular concept. The relations between CDFs describable using the CDF are as follows:

- version,
- derivation,
- decomposition, and
- interfacing.

Versioning supports several alternative descriptions to be given for the same concept. This gives a view of the CDF a form for describing design alternatives for a common software concept. Derivation relates a specific concept to its source concepts (Requirements R1 and R2). This relation can be used to record the design histories of software concepts. Thus a set of CDFs can be viewed as recording a design history as in Figures 5.11 and 5.13. Non-atomic concepts are related to their component concepts using the decomposition relation (Requirement R3). Thus a set of CDFs can be used to impose a hierarchical structure on a complex design description, see, for example, Figure 5.5. The description of interfaces within a set

of CDFs allows more detail to be added to the decomposition (Requirement R4). The means of connecting one concept to another or its environment is given through identification of interfaces, both internal and external; and these are related through the bindings listed in the CDF for each component concept, for example, see the CDF listed in Appendix D.

In order to record the above relations between software concepts, it may well be necessary for the domain analyst to do more than abstract existing software documentation. To some extent, this will depend on the level of detailed description attempted. Because the CDF was developed to support reuse of software concepts at various levels of abstraction, it may be used to record descriptions of concepts at various levels of abstraction and such descriptions may be found in the domain literature covering application software in general (Requirement R5). Through use of the CDF to describe both general application concepts and specific concepts via the above relations, a set of related CDFs can be developed which potentially support the reuser more effectively than isolated concept descriptions.

The effectiveness of the CDF in practice during the application studies when describing these relations considered above will be considered below. As explained in Chapter 5, the CDF development was used to populate the concept database of the PRESS through a series of domain studies. It is primarily this experience gained in application of the CDF that provides the basis for the subsequent discussion in this chapter.

6.2 Using the CDF in Practice

The application of the CDF in domain analysis remains a difficult task. Many major application software subfields lack standard references as noted in Chapter 2. This problem is compounded by the varying standard and level of existing documentation available in some domains, and by the diversity of languages used in software description.

Even though the software documentation studied in the steel domain was usually of a high standard, it was often not detailed enough. This was the case with respect to timing information. One aspect of software not explicitly covered by the CDF is timing. This was discussed with domain experts in steel as it is an important element in designing control systems. In the CDF, timing constraints could be recorded in Part 1 as part of the requirements under the Description of Purpose. Timing could also be addressed in Part 2 in the concept definition either formally or informally. However, in the steel domain studies, such timing information was rarely given in the documentation studied, and the case for developing a more detailed description of timing generally was considered beyond the scope of this research.

The diversity of languages used in the description of software presents a challenge to the domain analyst particularly in established application areas such as these studied here. The goal has been to record software concepts realised in existing software working from existing documentation and other available references. Although using the CDF did impose a uniform structure on the concept descriptions, no attempt was made to solve the problem of diverse languages at the lowest level. The CDF is a compromise between a common language, which would bring the overhead of translation, and simply recording existing descriptions without any restructuring. The advantage of imposing a uniform structure over the descriptions is that the reuser has only to become familiar with this new form when consulting the concept database to identify the parts and level of the CDF details that are of potential interest. While a common formal language could bring even greater benefits relieving the reusers from having to familiarize themselves with specific notations, the approach of translating existing software concept descriptions into a common language was ruled out because in the main domain studied (i.e. that of control systems in steel), there was no common formal language in use and to employ such a solution would not have been acceptable to the potential reusers participating in the project.

Application of the CDF does entail more than restructuring of existing documentation because the CDF also records the derivation relation if known and the decomposition relation if the concept is non-atomic. In addition, the domain analyst must

identify and classify the various interfacing concepts associated with a particular software concept.

It is the support that the CDF gives for recording these relations and interfaces that underlies the claim that the CDF may be viewed as an interconnection language, and in the applications of the CDF made here, these relations have been illustrated using sets of CDFs from both domains studied and reported on in Section 4.3 in Chapter 4 (see Figures 4.2, 4.3 and 4.4) and more extensively in Section 5.3 in Chapter 5 (see Figure 5.5).

These relations and interfaces may not be obvious from study of the existing documentation, and it may be necessary to consult other background documents describing software concepts of the particular domain. At this stage in the domain analysis, the identification of general system models is an important step as Chapters 4 and 5 demonstrate. Below in turn, the role of general systems models in supporting the description of concept derivations, concept decompositions and concept interfaces is discussed in more detail.

6.2.1 Recording Concept Derivations

The approach to domain analysis employed in software concept description has been to use a common form to structure all the descriptions of concepts - the CDF - irrespective of the domain studied and irrespective of the general or specific nature of the software concept. In applying the CDF, an attempt was made to record the most general concepts first although in the steel domain this proved more difficult than in the compiler domain. In both cases, the utility of having such general concepts to guide the studies was apparent. In both domains, general literature rather than existing software documentation was found to the best source of these general software concepts. This shows that here at least as used to record concept derivations, the CDF is more than a structured form recording abstracts of existing software documentation.

Experience in applying the CDF in the compiler domain has shown that where compilers in general have been described in the framework of common system models, CDFs recording these models provide insight to understanding the specific software concepts that are likely to found on examination of existing compilers. Determining the concept derivation of specific concepts was easier where CDFs for the more general concepts had already been filled in. In fact, the reference source used for this simple illustrative study presented the general concepts first and only later discussed actual compiler implementations. In studying the compiler domain, it was possible to draw on a large body of established literature on compiler construction distilled in a single reference source.

Likewise in the steel domain, there existed a large body of material which was relevant to understanding the software concepts employed in the design of control systems in steel production, although as pointed out in Chapter 5, in this case, the material was not available in standard textbooks. Only through study was it possible to determine the common themes and models underlying systems design in the steel domain. Again, the starting point was not existing software documentation.

In Chapter 5, particularly in the field studies of a specific system controlling a tandem mill, the work although based on existing documentation was greatly helped by the availability of a general reference source on tandem mill automation and other reference works, such as the so-called bible of steel - **The Making, Shaping and Treating of Steel** and the Williams report from the steel domain which enabled the author to deduce that a tandem mill was a form of cold rolling mill and obtain a general account of the control system associated with a cold rolling mill. This enabled a derivation to be traced from the general model of cold mill control through to tandem mill automation in general through finally to the specific concepts employed in the software for controlling the schedule of the tandem mill at the Peine-Salzgitter steel works as described in Figure 5.11 - Conceptual Derivations for Concepts in Cold Working Control Systems.

Given the utility of such general models, it is worthwhile investing effort in searching for these when initiating work in a new domain. If the domain is not mature enough

to have already given rise to the development of established models, it may not be particularly suitable for supporting concept reuse. However, the domain analysis required to develop system models may provide the necessary impetus to codify existing design expertise as a first step towards establishing concept reuse. Literature from the domain including software life cycle documents is the starting point for this work.

6.2.2 Recording Concept Decompositions

In all of the studies, the concept decompositions were readily available from the existing documentation or reference source. For example, in the ALP study, the system designers had employed a hierarchical decomposition which was reflected in the component naming found in the documentation. Some of the ABB offers studied included decomposition diagrams relating the main system components such as that given in Figure 5.7 - Decomposition of Hot Dip Galvanizing Line Process Control.

The general system models discussed above were also found to be a valuable guide to determining standard system decompositions to guide the identification of concept decompositions recorded in CDFs. All the CDFs developed from the Williams report were produced by systematically working through the report and the fact that CDFs for the more general concepts were produced first guided the subsequent work. As noted as Section 5.3.1, the CDFs developed from Chapter 5 of the Williams report provide a model for the CDF developed from Chapter 12 even though the control tasks are described for each of the subareas of control of the hot mill unit.

In the domains studied, although standard decompositions were available, it was still difficult to establish the specific concept decomposition from existing software documentation in some cases due to inconsistencies. For example, during the field studies at Salzgitter, the existing software documentation was found to be inconsistent. Further study of the code in consultation with a local control engineer was necessary.

More generally, where models of standard design decompositions are not available, in analysing the domain software, the domain analyst must attempt to abstract these from studying a number of specific systems. It may be difficult to recover the original designer's intentions. One approach assuming well-engineered software is available for study is to attempt to apply in retrospect a variety of software decomposition guidelines such as those applied in software development.

The classic paper on the principles of software decomposition is Parnas' *On the Criteria To Be Used in decomposing Systems into Modules* [111]. In this paper, Parnas had the remarkable insight that

> *a careful job of decomposition can result in considerable carryover of work from one project to another*

long before the concept of software reuse was fashionable. The historical design records will necessarily reflect specific system designs and also the historical development of the design. Such raw data is simply the starting point. A particular problem is that retrospectively working from the software realised as code may result in software concept decompositions that are too specific.

Even in the case where a standard design decomposition is available, a mismatch may arise between the decomposition found in the code and the decomposition expected in terms of application concepts anticipated from the standard design decomposition. To resolve this problem and establish the relations between the concept descriptions at various levels, it may be necessary to rationalise the original design by examining the designs of several related systems.

Parnas has also given good pointers on how to start with the task of rationalising design documentation [112]. The possibility of abstracting out the architecture of a system from its existing realisation has also been described by [75] although this thesis is advocating that the architecture is abstracted from several related systems to obtain general system models. An example of a general model described as a framework can be found in Basili [16]; here the basic design concepts of systems to

support component reuse have been abstracted out although the realisation that such a framework itself gives a basis for higher level reuse of designs goes unremarked. In the CAMP work, reuse of an architecture for a common Ada missile control system was supported but the work did not attempt to go beyond this towards a notion of generalised architectural concepts [12]. All of the work cited proved useful background for the guiding the analysis prior to recording a decomposition using the CDF.

In using the CDF where the focus is the description of potentially reusable software concepts abstracted from descriptions of existing software, the above work cited has been a useful source of software decomposition guidelines relevant when attempting to reverse engineer the system's component parts, but these need to be augmented by more basic conceptual analysis to determine the derivation relations between concepts. The guidelines of Maibaum and Turski outlined in Chapter 2 are relevant. These guidelines are of assistance in determining which concepts are fundamental to the system design or to use Maibaum and Turski's expression *descriptive-theory building process*; the results of which are expressed here with an incomplete set of CDFs at the end of Step Two of the domain analysis following the broad steps proposed by Prieto-Diaz of:

1. preparing the domain information,
2. analyzing the domain, and
3. producing the reusable workproducts.

In this context, it is helpful to distinguish between program understanding and application understanding within the process of domain analysis. In examining existing software for its reuse potential, reverse engineering of design components may be facilitated by applying software decomposition guidelines to the existing software to gain a higher level description of the software's components. It is also relevant to work from the domain concepts used to describe application systems in order to determine the underlying system models from which specific system

models have been derived. Ideally, the domain analyst carries out the application understanding first and then starts on the program understanding.

In the simple application of the CDF in the compiler domain analysis, this was possible, i.e. in the domain analysis, Step Two was performed before Step Three. In practice though this may not be possible. These two steps may need to be performed out of sequence or together. It may be that studies of specific software to support its reuse take place before any system models have been formulated, or the system models found may need to be developed and studies of specific software systems are undertaken to provide the foundation for the development of the general system models. In the large application of the CDF described in the previous chapter, the domain analysis progressed as an iteration of repeating steps two and three, sometimes working from specific software documentation including source code and sometimes from more general system models found in the domain literature.

In the approach developed here, the CDF was used to record the results of both Steps Two and Three of domain analysis. In this approach, general system models recorded during Step Two have provided the basis for the recording of more specific concepts in step three. The CDF can be used to describe sets of related concepts and together these may be consolidated into design frameworks in particular domains. By forming the sets of general CDFs into design frameworks, the work here suggests that such frameworks have a role in both guiding further domain analysis as well as supporting the potential reuser in understanding the context of individual CDFs in relation to each other.

6.2.3 Recording Concept Interfacing

The general system models were also found to relevant to the working of identifying concept interfaces. This was especially found to be the case in the studies of software concepts in steel mill control systems where the four level model employed to describe such systems overall was used to locate specific software concepts and their external interfaces which typically exist at the boundaries between these levels.

The arbitrary interfacing of software concepts from a variety of domains has not been addressed by this thesis, and even within the limited domains studied, the arbitrary interfacing of software concepts at the lower levels of description, such as code level, is problematic. In the Steel Production domain, reuse at the code level is not a primary concern, and this thesis has not presented any significant solutions to this problem. It is acknowledged that the CDF is weak with respect to addressing one requirement highlighted in the AISE study - that of need for standard interconnection mechanisms. This could be addressed by giving more consideration to the description of interfacing concepts. In the steel industry, a partial solution has come from the adoption of industrial networking standards such as the Hoogovens Data Network mentioned in Chapter 5 and employed in the demonstration system offer.

In the more general case, the author's view is that further work on improved description of interfaces will only make the problems of interfacing more clear and not lead to significant advances unless the emphasis is focussed on improving the interface descriptions at the conceptual level using higher level design descriptions than code. Here Weber has pointed the way with his proposals discussed in Chapter 3. The CDF is only a small step in this direction, and in applying the CDF to describe concepts realised in existing software based on existing software documentation, it was found that such higher level descriptions when available were not given formally. As the software industry moves towards a greater use of formal methods in the early stages of design, this situation can be expected to improve. This discussion is developed in more detail in the following section.

6.3 The Need for a More Standardised and Formalised Description of Software

The CDF falls open to the observation that it is unlikely to succeed in widespread usage unless it is adopted by the software industry. Although the practicality of

using the CDF to describe software concepts in an industrial application has been demonstrated, it is not claimed that the CDF as it stands presents the solution to the industry's need for a more standardised and formalised form for describing software. Here there is a conflict within the industry which works against the adoption of any one form even given that the advantages of doing so are great as envisaged by Weber's proposals discussed in Chapter 3.

For example, within the company ABB, the various business divisions introduced to the CDF were conscious of the effort required to develop a database of CDFs based on their existing software documentation and to maintain it for a long enough period to obtain the benefits of reuse and were resistant to commiting their divisions to the use of any particular form; they felt that each business division should have the flexibility to define an overall form relevant for their own software applications [1]. Given that the CDF is a quite minimal form for structuring software documentation, it goes some way towards meeting this requirement for flexibility and as such accommodates various levels of software descriptions, but it is still recognised that considerable effort is required to develop and maintain sets of CDFs for any particular domain. It is the author's belief that the move to a more standardised and formalised description of software within the industry can only be achieved gradually and must rely on a better formal understanding of software design concepts than is present in many domains.

The form developed - the CDF - merely imposes a structure on software concept descriptions rather like Z schemas. The developers of the specification language Z built on the notation of set theory and logic introducing the schema form as a means of structuring specifications. The CDF allows any existing software documentation to be structured. It provides a filtering mechanism through which the leveled description of the software's concepts can be achieved.

A very deep question concerning this work is: can design be completely formalised? The author's view is that this is an anti-progressive goal. In reuse research, the goal should be to describe and where possible formalise existing concepts so that these can be employed in the development of new concepts. It is the author's belief that while

it is best to formalise, where possible, existing concepts for reuse, it is unproductive to exclude informal accounts of existing concepts from the reuse process. Both formal and informal accounts have a role; and developing new concepts or simply better versions of old concepts is an ever present challenge to the designers of software systems.

In proposing the CDF, a structure enabling both derivations of concepts and decompositions of concepts to be described at various levels of abstraction has been established. A major deficiency of this work is that a CDF calculus is missing to aid reusers. Here there is potential for developing this work along the lines of the research of Henderson and Warboys discussed in Chapter 3.

The history of the development of schema calculus briefly outlined by Woodcock in [173] gives some insight into how formalisation can be added to a structuring notation. According to his account, originally the schema notation was simply a convenient shorthand with macro-like expansion of names and the calculus was developed later.

In this work, the author has concentrated on ensuring that the structure of designs is recorded and made available for reuse as well as recording details of reusable components. The importance of structure in design is the final justification given for this work on the CDF; in what follows, the case of reusing design structures is made and the support that the CDF provides for structuring concepts is evaluated.

6.4 The CDF's Support for Reuse of Known Design Structures

Building on McDermid's insight that most improvements in our technology for large scale software system development have depended upon the finding of improved abstractions or structuring techniques for describing software [99], an examination of the role that structuring plays in the design process is relevant in evaluating the

CDF with respect to its support for reuse of known design structures.

The CDF as proposed specifically aims to support software designers in reusing known levels of abstraction and structures if these already exist in a relevant application domain.

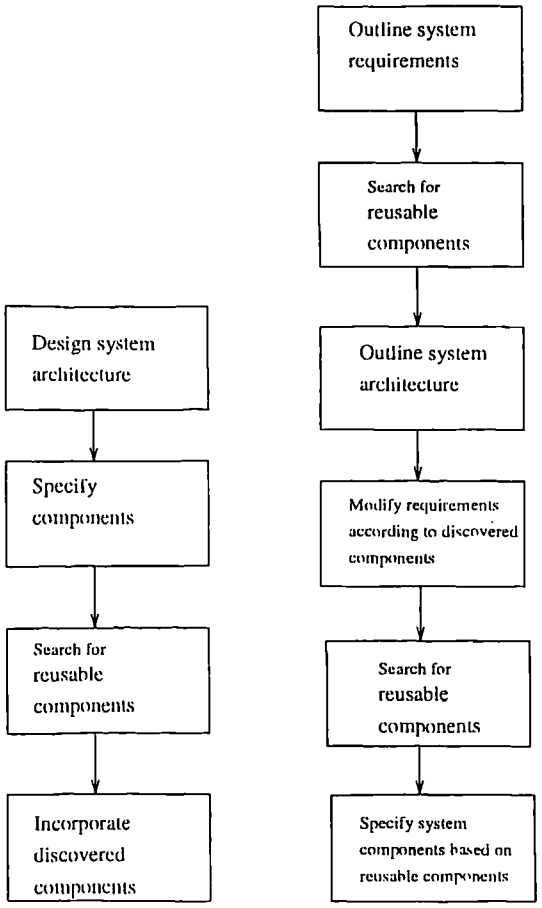
The three essential stages of design have been identified by Jones [85] as:

- *breaking the problem into pieces* (Analysis)
- *putting the pieces together in a new way* (Synthesis)
- *testing to discover the consequences of putting the new arrangement into practice* (Evaluation).

In the context of system design, Jones refers to these stages as: divergence, transformation and convergence. The successful understanding achieved here is very much dependent on having sufficient information and experience on which to draw. For example, in the studies reported here, through a process of domain analysis, resources have been built up for the designer to use as the requirements texts are broken down into terms and understanding is assisted by mapping these terms onto known concepts using the thesaurus and concept database. Since the mapping achieved is rarely one-to-one, Jones' term, divergence, is particularly appropriate for describing this element of the design process. The next stage builds on the results of the understanding gained, bringing order to the divergent interpretations; so that finally the designer is able to converge on a single design to be developed in detail. Jones points out that with increasing automation of design, detailed development is likely to become the bit that people do not do. Certainly this is a part of the promise of reuse: to free the designer from unnecessary re-development of established concepts, and the work here specifically seeks to address reuse of established structures in design.

Early on in current system design practice, informal architectural sketches are employed to describe the over-all structure of the proposed system; and then once the

main component concepts have been identified within this context, detailed design or reuse of existing software components may proceed; for example, Sommerville characterises development with reuse as shown as Figure 6.1 - Models of Component Reuse - reproduced from [147].



Development with reuse contrasted with Reuse-driven development (taken from (Sommerville 1992)).

Figure 6.1: Models of Component Reuse

In both of these approaches, the designer works from a library of reusable components, but the overall system architecture is not considered as an aspect of design where reuse is possible. More than the component approach to design illustrated above is needed to support the designer at the early stage of overall system design if reuse is to take place at this point. This is where the engineer could benefit from a library of design decompositions made for existing systems.

In addition, if these design decompositions can be related within a framework which clarifies their conceptual basis at the appropriate level and gives the designer insight

into why these decompositions were made, their reuse value is enhanced. At the most abstract, such insight will take the form of design principles; at a less abstract level, perhaps only guidelines will be available.

To support design reuse at this level, software engineers might do well to learn from the example of civil engineers. The Architecture and Engineering Performance Information Center was established at the University of Maryland to maintain a database of structural design experience and material usage experience (these are the analogues of architectural design concepts and component concepts for civil engineering) [115]. The proposed design frameworks built out of CFDs of this thesis are a meta-structure over architectural design concepts and component concepts formed. They make clear the relations amongst generic and specific forms of both these types of concepts. The framework for software concepts in steel given in Figure 5.15 provides an overview of concepts studied and their relations.

This does not mean that libraries of reusable components are unimportant. They have a role, but much later on in the design process. If the engineer sketches the architecture of the design and then looks for components, by this time through poor initial decomposition, it may be too late to help. Randell has argued that poor design structuring, i.e. decomposition, can doom a system [131]. If good structuring by expert designers is not made explicit and recorded, then the reuse of design structures is held back.

Another way of justifying this approach is to consider that in order to stock the library with reusable components, existing systems were analysed and decomposed in order to identify suitable reusable components. Rather than throw away the results of such analysis, the structures found in decomposition should be abstracted out and made available for reuse. This is possible using the CDF. And if a designer chooses to reuse a known decomposition, through the other entries provided by the CDF, details concerning its component parts can be followed up and also considered for reuse.

Inspection of known decompositions at this point may also lead to the identification

of unstated or assumed requirements. In the demonstration described in Chapter 5, the decompositions found in the retrieved CDFs suggested that a component for material tracking was required even though this was not mentioned in the customer requirements.

In the software design process, it is important to distinguish between architectural concepts which are relevant to the system structure and component concepts which reflect the constituent parts of the systems [28]. This distinction is made by Shaw in terms of *higher-level* abstractions (i.e. architecture level) and *lower-level* abstractions (e.g. algorithms and data structures) [140]. The CDF provides the possibility of recording both sorts of abstractions for subsequent design-with-reuse.

Moreover, the use of layers in system design is a well understood approach to system organisation; a classic example of this can be found in the ISO Reference Model of Open Systems Interconnection (OSI) [176]. Here Zimmermann gives a more thorough articulation of the principles underlying the use of layering. Three main principles, taken from the account given in [151], are as follows:

1. Create layers where a different level of abstraction is required;
2. Choose layer boundaries to minimise information flow across the interfaces;
3. Use a large enough number of layers to ensure that distinct functions are not thrown together in the same layer out of necessity, and a small enough number to ensure that the architecture of the design does not become unwieldy.

In Chapter 5, it was shown how the ISO 7 Layer Model could be interpreted as framework relating existing network architectures.

These principles give the designer insight into how to make use of layering during the conceptualisation of the design structure. They are relevant to the designer when conceptualising the architecture of the system. This layering of the system design is a fundamental step in conceptualisation; there is more to conceptual design than

simply dividing the system into component parts. To support the designer at this stage, more is required than simply libraries of software components descriptions.

Above various approaches to design and the role that structuring plays in these have been examined. These form a rationale for to the development of CDF sets providing design frameworks to support reuse of known design structures. In order for reuse to take place during conceptual design, a need for supporting the designer at this stage with known conceptual structures and structuring principles has been established. In Chapter 5, through the domain studies carried out in steel, it was shown how a framework for relating known design structures was identified and how this assisted both the design-for-reuse and also has a role in supporting design-with-reuse.

6.4.1 Further Considerations Regarding the Use of Frameworks in Design

In characterising the design process, Guindon remarks as follows:

> *... because design problems are ill-structured, the design process cannot just be the retrieval of known solutions, even in experts. The novelty in design and the incompletely specified requirements force even expert designers to punctuate retrieval of known solutions with the inference of new requirements, the recognition of partial solutions at various levels of abstraction, and the creation of new solutions.* [71].

Following Newell and Nii, Guindon characterises design as a problem solvable from the application of knowledge in the form of empirical associations or rules derived from past experience. Here from this point of view, it can be seen that reuse is already assumed to be part of the repertoire of the expert designer albeit reuse of the individual's own experience. A clear role for design frameworks to record past design expertise can be made in the light of Guindon's study. Guindon argues that retrieval of potential solutions is not sufficient as the designer is actively engaged throughout

the process revising and adding to requirements if necessary, and attempting to identify partial solutions. His study is limited to designers attempting to understand and elaborate requirements, i.e. Phase 1 (Clarification of the Task) in Pahl and Beitz characterisation of the design process [110]. An important point made by Guindon is that this phase in software design cannot easily be separated out as problem understanding and structuring were found to be interleaved with solving the problem throughout the design sessions studied.

It is clear from the account of the lift controller design histories given in Guindon that in the case of the second designer studied the ability to recall and use a high level design schema that relates what in the terminology of Pahl and Beitz are the classifying criteria to the solution characteristics was decisive in guiding his approach to the design task. In this case, the criteria were those generally found in a resource allocation system: multiple clients, multiple servers, limited resources, asynchronous service requests and routing/scheduling optimisation. In the schema used, these mapped onto a design decomposition involving three main characteristics: control, communications and scheduling. In considering this case, once the designer had recalled the schema, it is clear that it was the primary determinant that led to his effective solution of the task. As Guindon remarks:

> *Design schemas are one source of knowledge that powerfully constrain the search for a solution.*

Further, he concludes that top down processing induced by the design schema contributed to a systematic design process. The way in which the second designer was able to employ a high level design schema supports the envisaged use of the design framework for steel production. Guindon cites examples of design schemas described by Lubars in [96]; here the schemas given involve very high level abstracting out of design solutions common to several application domains. Lubars identifies the encoding of reusable design information into design schemas as one of the key aspects of automated design systems. In this work on a design framework for steel production, there has not been any aspiration to achieve automation of the design

process. Nevertheless, abstraction within the domain has proven a useful means of developing a framework supporting reuse of designs on a smaller scale.

From these discussions, it can concluded that it is helpful to advocate that the initial search of the designer is guided by an understanding based on giving structure, possibly via abstraction levels, to the design requirements. This is where the capability of drawing on existing design frameworks identified through similar requirements comes into the design process. Compare here the steps found in Pahl and Beitz characterisation of conceptual design where the designer is advised to abstract away the details given in the requirements in order to identify the essential problem and then to establish the functional structure of the solution in abstract terms initially using these as the basis for identification of known functions to apply in the design. With the development of design frameworks, the designer is offered a basis for identifying known structures, and thus, reuse is promoted earlier on in the design process. The answer to the question: what is the role of these frameworks relating system models in design?, is as follows: they are the carriers of very high level design concepts. Inspection of known decompositions at this point may also lead to the identification of unstated or assumed requirements.

Ths role identified for design frameworks has not yet been investigated extensively with the framework for steel production presented here; in the context of the Practitioner project, it formed a contribution to the development of demonstrations of the PRESS in the preparation of offers which utilized the framework. One of these has been described in Section 5.4 of Chapter 5. In the demonstration described here, it was possible to locate the CDFs retrieved within the steel design framework at the appropriate level of control and area of control as required by the customer. By inspecting the decompositions found in these CDFs, it was found that a component for material tracking was required even though this was not mentioned in the customer requirements.

Empirical studies of designers are problematic and were not possible within the timescale of this research; however, presentation of this work to practitioners in the domain has been favourably received and this has strengthened the view that

the framework developed has a real role to play in actual design. In addition, the practice of using models in control system design has been noted and the CDF sets developed clearly have a role relating system modelling concepts in the steel domain as described in Chapter 5.

With the work reported in Chapter 5, the key factor in the success of reusability cited in [88] and reiterated here has been further established:

> *Reusability can be achieved more successfully within a narrow and well analysed domain. This enhances the possibilities of identifying functional commonalities in the application software and will make the development and cataloguing of components easier.*

The final study reported within the Steel Production domain has enabled the earlier domain studies using the CDF to be built upon and a framework for guiding further analysis and also for relating the design concepts available for reuse in the development of future steel plant control systems to be refined.

A speculative conclusion of this work turns on the utility of abstraction in design. In the discussions of design models in software reuse, the importance of abstraction has been identified as the keystone to successful software reuse [93, 64, 168, 150] and others.

More generally, in their account of a systematic approach to engineering design, Pahl and Beitz advocate that initially the designer abstract from the specifics given in the requirements to the general case in order to achieve a less constrained viewpoint from which to attack the design problem conceptually [110]. In the context of software engineering, Shaw's remarks on the need for higher levels of abstraction are perhaps the real inspiration behind this work [141, 142]; she brings out quite clearly how intellectual progress is purchased through the difficult process of abstraction citing examples from other engineering areas.

In contrast, it has long been known that the strategy of divide-and-conquer is a successful approach to attacking intellectual problems such as design. This has led many to argue that hierarchical decomposition of design from the standpoint of design tasks attacking sub-problems using known plans is the way to bring past experience to bear. For example, in a general study of design taken from an Artificial Intelligence standpoint, a classification of design is given in terms of known problem decompositions [39].

Through a vertical decomposition based on levels of abstraction coupled with a horizontal decomposition into a whole-part hierarchy, a design framework such as the one presented here based on domain analysis using the CDF allows these two approaches to be combined and offers the designer a more powerful, conceptual basis for reusing designs.

Sets of CDFs are required to express and support abstraction within the design's architecture. This is where the CDF as a structuring form to sets of related concepts can be applied. The level of abstraction used to describe design frameworks must allow the generic framework of classes of common applications to be expressed; it must also be possible to express possible interconnections and dependencies among the various parts of the application. The description of interconnections and dependencies is covered by existing interconnection language developments and as has been shown is supported by the CDF. Within a framework, it must be possible to record various derivations of architectural concepts. For this reason, the abstraction process has been extended over sets of related CDFs describing application concepts in a particular domain.

The development of a design framework may require that field work is undertaken to record design sketches and decisions which once captured are the raw material out of which the design frameworks will be abstracted. The CDF already described has been used to study applications in the domain of Steel Production. As a result of this work, it was possible to formulate general models of an industrial control system that were useful to production engineers responding to calls for tender on new developments as was shown in the demonstration discussed in Chapter 5.

The novelty of this approach lies in the abstraction of designs in the large and the proposal that the design framework be seen as a means of relating such architectural designs, both specific and generic, and as the vehicle for employing them in future system developments. Design with reusable components presents new challenges to those attempting to support the design process. Large scale reuse requires a larger view of design than that supporting component reuse; it is necessary to incorporate reuse of architectural designs of systems giving a context for component reuse. Here an outline of how the reuse of CDFs recording known design structures within design frameworks addresses this aspect of system design has been made.

Frameworks proposed in this thesis are very abstract and unlike those that have been proposed in Object-Oriented reuse research, such as that discussed in [3, 171]. These frameworks are very much less abstract and are intended to be realised within some object-oriented programming support system. The concept of framework in this thesis is more abstract as it is aiming to provided a means within which various generic application concepts can be related for a specific domain. The framework concept here is in some senses narrower than that of Object Oriented researcher as there has been no attempt to abstract out a generic framework to be used as the basis of applications across a number of domains.

One of the source concepts for design frameworks has been the use of reference models in the development of communications standards as discussed in Chapter 5.

6.5 Concluding Remarks

By placing emphasis on determining how software concepts have been derived and how they can be decomposed, this work has brought out the need to better understand and describe these relations amongst concepts in existing software. Although the CDF provides a structure for describing sets of concepts and these relationships amongst them, it offers no specific semantics for these relationships. In its present form, any semantics of the concepts is provided by the underlying semantics of the

existing software documentation and in the industrial domain studies, a high degree of formalisation was not found. Nevertheless, the structuring of descriptions contained in Chapter 5 was found to be useful to domain experts consulted at both ABB and Peine-Salzgitter.

It is the author's view that this lack of formalisation does not preclude informal descriptions of the structure of system concepts for the purposes of reuse. The work reported here constitutes a valuable first step towards a better foundation of design based on improved understanding of the software concepts found in the existing software of the domains studied even though the concept description was not formal.

Getting the structure in place, albeit informally, sets the context for describing the component concepts in more detail. These need to be described in detail before more formal description of the structural concepts can be tackled. So the steps two and three of domain analysis may need to repeated until the convergence of descriptions at an appropriate level of formalization is achieved.

In the more detailed application of the CDF in the Steel Production domain, this process was illustrated by the successive studies made culminating in the final study which consolidated the architectural concepts through the development a design framework. The framework developed allowed the component concepts already studied to be located within the levels of control and areas of control identified and also allows areas where further work in domain analysis at the step three level is lacking to be identified.

Chapter 7

Appraisal of Research

7.1 Contribution of this work to Software Engineering

The major contribution of this thesis is a systematic development of the Concept Description Form (CDF) and its application to support both design-for-reuse and design-with-reuse. The goal of this development has been to improve the reuse of design level concepts in the engineering of new software systems. The CDF has been applied in a number of studies of software designs in the domain of Steel Production. Existing domain models were unified using concept descriptions obtained with the CDF into a single framework, and this framework was further developed during the course of the research. The role of such frameworks in the reuse activity of domain analysis has been demonstrated. The usage of sets of CDFs to record design histories has been explored. A demonstration of how the CDFs from the steel domain could be reused during offer preparation has been made, and a favourable assessment of the CDFs and associated framework developed has been made in conjunction with domain experts. The application of design frameworks in the design process, particularly during the phases of Cognition and Conceptualisation. has been given

consideration although not demonstrated.

The area addressed by this work is the engineering of software systems supported by the reuse of software concepts. The primary interest is in supporting the software engineer early on in the development process with the provision of architectural concepts which provide a context for component reuse. The CDF has been proposed as means of recording and reusing existing system designs. The rationale for this approach has been developed by studying its application in the domain of Steel Production with the overall goal of improving software system engineering and contributing to a better understanding of Design Theory in a modest way.

This research has contributed to the development of a more systematic approach to domain analysis based on the CDF to support the reuse of software concepts. Largely, the research has been concerned with representational issues and understanding the role of reuse in the process of design. There have been two main results of this work as practically realised within the Practitioner project:

1. Population of the Practitioner REuse Support System (PRESS) with software concepts from the Steel Production domain; and
2. Demonstration of the PRESS with this material in the process of offer preparation.

This work has been supported by research into the theory of design with the goal of formulating a better understanding of design-for-reuse and design-with-reuse in the context of software system engineering.

Preliminary studies using literature from the Steel Production domain and material from Asea Brown Boveri's metallurgy business division were extended by field studies of software systems at the Peine-Salzgitter Stahlwerke. Work on design theory was further developed by studies of work developed by Pahl and Beitz in Germany under the heading of Construction Theory. Bringing these two threads together, a generic application architecture for the Steel Production domain was developed, thus codifying existing designs within a design framework of CDFs for steel software

concepts which became the basis for case studies in offer preparation within ABB. As a result of this work, the further consideration was given to the role of design frameworks in general.

These studies of the designs in the Steel Production domain have been both historical and intellectual; these studies have underpinned the classification of designs in the domain and also the development of a better understanding of how design-with-reuse could work in this domain. In the latter case, it was found useful to draw on more general studies classifying approaches to design.

7.2 Implications of this Work for Design Theory

In this section, consideration will be given to the more general applicability of the CDF and associated design frameworks. Here consideration will given to the implications of this research for design theory and methodology in general. Many of the problems addressed by the development and application of the CDF and design frameworks are not limited to systems development by software engineers; they are manifest in design in general. Here a brief account of relevant work in design theory is given, and parallels are drawn between this work and that supporting this thesis.

Design Theory brings together results from the engineering sciences and the practice of engineering, from systems theory, computer science and cognitive science. It examines the role that knowledge representation, mathematics, and computation play in design from both a practical and theoretical standpoint. The National Science Foundation study on Design Theory and Methodology defined design theory as:

> *systematic statements of principles and experimentally verified relationships that explain the design process and provide the fundamental understanding necessary to create a useful methodology for design.*

Where design methodology is

> *the collection of procedures, tools and techniques that the designer can use in applying design theory to design.*

(See [130]).

A distinction can be made between observational studies of complex phenomena yielding descriptions forming the basis of descriptive theories of design, and theoretical studies to develop formal models of the design process which can be used to structure and control the complexity of the design process, more prescriptive theories of design. Each approach has the potential to usefully inform the other and to contribute to the improvement of design practice. If the human activity of design is itself amenable to be being designed, then the possibility exists to develop new theories of design and radically change the nature of existing design practice.

The fact that an approach is prescriptive does not rule out its having empirical grounds for its theoretical basis. Both approaches look for empirical grounds to support their theory either directly or indirectly. The prescriptive approaches seek to give a theory to the process of design by giving reasons why design should proceed in a certain way and not others; and they propose that experimental evidence be provided to support each case. With respect to descriptive approaches, these look for confirmation from practical studies and cite experimental evidence to support the predictive power of their explanations. Both approaches may draw on experimental results from other disciplines such as cognitive science to indirectly support their case.

In an interesting study by Newsome and Spillers, results from psychological studies of problem solvers are presented [104]. From these, requirements for tools supporting conceptual design are derived. This approach illustrates how practical results from cognitive science studying current design practice can be brought to bear in consideration of how design ought to be supported. This study also identified three characteristics of the way that experts approach problem solving as follows:

- breadth-first approach,
- use of abstraction representation, and
- expanded memory of problem-related information.

It appears that experts in computer programming tend to take a breadth first approach in contrast to novices who use a depth first approach. These differences were found to be most marked as the complexity of the problem increased. These results were reported in [13].

Experts tend to use abstract representations or patterns in problem solving - results from computer programming, chess, Go and electronics support this. Novices in these areas concentrate on smaller, more concrete features of the problem.

Experts have a quicker grasp of problems; this is illustrated by their capacity to remember more of the problem related information. For example, experts in chess are able to reconstruct board layouts to a much greater extent than novices; the results quoted are greater than twenty pieces for experts compared to four or five for novices. Similar results were obtained for computer programmers recalling programs. One might hypothesise that this expanded memory is due to high level encodings of the problem related information.

As Newsome and Spiller point out, these three characteristics are not independent, and it is the second, the ability to utilize abstract representations, that underlies the both the first and third. The conclusions drawn from these characteristics with respect to CAD tools for expert designers are as follows:

> From the first characteristic, it is concluded that immediate closure of design subtasks is not necessary nor should it be forced on the designer. The CAD system should allow the designer to formulate rough global sketches of solutions prior to fixing any details in depth.
>
> From the second, it is concluded that only support for the most basic, or prototypical, representations are needed initially. This allows the de-

signer to use abstract patterns to shape the solution until greater specificity is desirable. In addition, any prompting queries from the CAD system should be of an abstract nature, e.g. clarifying relations among parts rather than prompting for specific details of parts.

From the third, it is concluded that the system should present the designer with global representations of information omitting details (the expert designer can fill these in when required).

The studies reported above are from domains where there already exist quite well-known and developed representations, i.e. chess and computer programming. In computer programming, there are a number of languages at varying degrees of abstraction that the designer may employ. This raises the question: What role do these abstract representations play in communication of designs? In particular, one may wish to determine whether or not certain levels of abstraction are more useful than others.

The engineer, Bucciarelli, has studied engineering design as a participant observer. In considering designing as a fundamentally social process, the work by Bucciarelli consists of exploring the nature of design discourse through which designers communicate under conditions of sustained uncertainty and ambiguity [41, 138]. He has identified the importance of multiple representations of objects-in-the-making, and has shown how multiple representations across the design discourse can both facilitate and inhibit convergence of the description of an artifact.

These findings argue for investigations into the appropriate levels of abstraction in any particular field of design discourse, to ensure that multiple representations are an aid rather than an obstacle to the design process. The work in support of this thesis on the CDF and design frameworks is a first step towards a means whereby the results of such investigations can be expressed.

The representations used by designers and supported by methods and tools must provide a sufficient and consistent vocabulary and grammar to enable effective communication at the requisite levels without placing unnecessary restrictions on the de-

signer's expression of ideas. A restrictive or biased design language limits the design possibilities. The designer should employ the most appropriate means of expression within any particular design exercise context - where one means is not self-evidently the most appropriate, a variety may be required to cover different aspects of the design. These points seem obvious. and yet there are always new prophets arriving on the scene advocating new design languages and new approaches to design without any supporting evidence. It is possible critically to evaluate a language; and to determine its properties, especially the scope of its application. Simon called for such efforts as a part of his programme of topics for instruction in the science of design over twenty years ago [144]. It is recognised that current software engineering practice necessitates that a variety of design notations be employed depending on the level and nature of the design concepts being described [98]. In this thesis, it has been shown how these may be related by employing a standard form for descriptions.

Of course, in reality, where a variety of professionals participate in the development of a design, consideration must be given to their varied interests, strategies of representation and different languages. In reality, such "layering" of meanings is central to design discourse. And part of the research in design theory must address the provision of conceptual structures in which these experts' structures can be structured. It is here that the work on design frameworks based on the application of the CDF in this thesis makes a modest contribution to design theory.

These considerations have given rise to several topics for future research discussed in the following section.

7.3 Directions for Future Research

There are two main directions in which the research described here will be carried forward:

1. further research in software engineering investigating how the CDF and design frameworks could be employed in software maintenance as well as reuse, and

2. further research in general engineering design investigating the general applicability of the design frameworks and the CDF.

A broader perspective of software engineering research now views the whole of the software life cycle as inherently encompassing both software reuse and maintenance, see, for example, the research agenda proposed by the Computer Science and Technology Board reported in the March 1990 issue of the CACM [51]. A long-term action identified as part of this agenda is the building of a unifying model for software system development. This research action is motivated by a fear that many of the large computer-based systems on which our society depends are becoming unmaintainable, and by a pressing need to improve our understanding of how to create and maintain large and complex software systems. Through the reuse of software concepts, the designer is able to build new systems on established conceptual foundations while still taking advantage of improved technology and techniques in the implementation of the new system. In addition, an understanding of the software concepts employed in the construction of a system also provides a firm basis for its maintenance, particularly where, in the form of perfective maintenance, an evolutionary approach is taken.

Futhermore, large software systems have evolved to the point where many organisations both in industry and commerce could not operate effectively without them. It is important to ensure that the knowledge and experience of those commissioning, developing and maintaining such systems takes a recorded form open to scrutiny for managerial as well as technical evaluation and appraisal during their operational lifetime. Such understanding of software must be recorded at appropriate levels to ensure its accessibility to all those within an organisation who need to rely on it throughout its lifetime. The use of the CDF to build design frameworks will be investigated for this purpose. Although the initial focus in development has been primarily on describing software concepts at the higher levels of abstraction required to facilitate their reuse during the conceptual phase of design, this further research

will consider its potential for recording levels of understanding more appropriate for maintenance of existing software.

The other main direction in which this research will be carried forward is to investigate the general applicability of the CDF and design frameworks in the context of general engineering design. Here the above considerations relating this research to design theory are relevant. Two main research actions have been identified:

- to investigate the application of the CDF and design frameworks supporting design-for-reuse and design-with-reuse in other fields of engineering giving consideration to implementation as well as conceptual design with an aim to refining and generalising the concept of design frameworks;
- to employ design frameworks to support the multilevel communication amongst a team of designers working on a large development project (concurrently and over time) as the design is evolved.

In the first action, the future development can build on established design handbooks in other areas of engineering. As many engineering design teams are interdisciplinary, combining concepts in a multilevel framework will form the basis of the second research action. In the latter case, development of design frameworks to support Computer Supported Cooperative Working in the area of engineering design will be undertaken building on the experience gained here applying the CDF in software engineering.

Two much wider issues related to this research are:

- the role of formality in design (c.f. Alexander's recantation found in the Preface to the Paperback Edition of [9], and
- the wider implications of design such as determining whether or not a design is in harmony with the rest of the world (considered in [9, 63, 170, 110]).

The role of formality in design is a vexed question. The crux of the issue would seem to be that if the process of design can be formalised and carried out automatically then it would cease to be intellectually challenging. Alexander's view is that blindly following any design method is futile. This thesis has not directly challenged the concept of automating the process of design, but a further development of this work would be to confront this concept and establish its futility.

The latter issue is a problem for all professional designers. In the design of software, the usual focus is on getting the specification to meet the requirements and the system to meet the specification (i.e. validation and verification), but *little attention* has been given to the broader context and implications of design, for example, see Winograd and Flores, and Floyd for a discussion of this with respect to software. These authors' work argues that designers have a responsibility to consider the ramifications of their work on society at large.

While this thesis does not expressly take up either of these issues, the approach of design frameworks with its emphasis on understanding the broader context of a design could provide a practical means of beginning to put consideration of such issues on the design agenda of practising software engineers.

7.4 Summary of Thesis

Chapter 1 set the context for this research within the field of Computer Science known as Software Engineering and, in particular, within the area of Software Reuse. In Chapter 2, research directions and outstanding issues in software reuse were summarized. The need for work on improved methods of software description to support reuse was discussed; and the specific concerns to be addressed by this thesis were outlined. Chapter 2 reviewed specific reuse research and gave an account of the Practitioner Project setting the context within which this research was undertaken.

The heart of the thesis can be found in Chapters 3, 4 and 5. The thesis has sought

here to fulfill four main aims; they are as follows:

- to develop further a generic form for describing software concepts;
- to establish the adequacy of the representational form for describing reusable software concepts;
- to employ the form in small and large scale applications and evaluate the form as a means of supporting software concept reuse through populating a design concept database;
- to gain an understanding of the inherent role that concept reuse plays in design and to make this more explicit through providing a means of recording design frameworks.

Chapter 3 of this thesis discussed in greater detail the form of software concept descriptions developed to support software concept reuse. The requirements for a language to support software concept reuse were elaborated and the form of software concept descriptions, the CDF, developed as part of this research was discussed in the light of these requirements in order to establish its adequacy as a means of supporting software concept reuse. It is acknowledged that the form developed is a minimal form.

Fulfillment of the third aim is based on the studies made in Chapter 4 of this thesis supported by a more thorough going application of the CDF in the development of a design framework and analysis of its usage in practice found in Chapter 5. In Chapter 5, various studies in the domain of Steel Production were reported. This work resulted a better understanding of the domain's software concepts found in the designs studied and recorded in sets of CDFs. The work also resulted in a realisation of the potential for design reuse in this domain; CDFs describing general concepts were used to guide the development of CDFs describing concepts from specific systems, and a demonstration of concept reuse based on a set of CDFs during offer preparation was made. Work in the steel domain provided further opportunities to apply the CDF in populating the PRESS; a major result of this

work has been the development of a more comprehensive design framework for the domain based on sets of CDFs. The role of this framework in design within the steel domain has been investigated.

Chapter 6 evaluated the CDF and its application to support reuse and to establish the longer-term prospects for the CDF and design frameworks. Consideration of the role of such frameworks in design formed the final part of this chapter.

Fulfillment of the final aim is dependent on the comprehensibility of this thesis taken as a whole. Throughout, the role that concept reuse plays in design has been considered; and with the development of the CDF, a means of recording descriptions of concepts used in design has been provided. By establishing relations between concepts and explicitly recording these in sets of CDFs, it is possible to develop a framework of design concepts to support their reuse.

7.5 Conclusions

The most general goal of this work has been to understand and improve the processes as well as the products of Software Engineering; in particular, to understand the process of reusing existing software concepts in the development of new systems. A useful direction to this research has resulted from applying general engineering design principles from Construction Theory to the engineering of software. Two aspects of design have been examined:

1. Design-for-Reuse (Population of the PRESS) and
2. Design-with-Reuse (Exploitation or usage of the PRESS).

With respect to the products of software engineering, it has been found useful to distinguish between specific and generic software concepts and between structural concepts such as the ISO 7 Layer Model of OSI and component concepts such as a

parser in a compiler. This work has concentrated on the refinement of a canonical form for the description of software concepts, the CDF.

The research to support this thesis carried out within the Practitioner project has had a two fold aim; developing a systematic understanding of the procedures necessary for populating the PRESS, and developing a systematic understanding of the process of using the PRESS in the creation of new designs. This is the context within which the approach of design frameworks based on the CDF has been developed.

An examination of the role that system structure descriptions play in high level design reuse has been carried out through the development of the concept of design frameworks. Design frameworks can be seen as an attempt to induce a history on a set of designs, to establish a design tradition, and to erode the possibility of change for its own sake, i.e. the needless adaptation which destabilises a design tradition.

It has been shown how design can be supported from known concepts using multi-level frameworks populated with sets of CDFs. Frameworks allow designers to relate CDFS which describe common application structures or functional structures, that is forms of system architecture. A software concept may be decomposed into its component parts, i.e. using whole-part relationships, or it may be decomposed into levels of abstraction giving descriptions of the whole system from various viewpoints. These decompositions allow designers to construct two dimensional frameworks within which various design concepts can be related.

Although throughout the research reported here, the focus has been on the representation of software concepts, the scope for wider application of these results is recognised. This work has implications for research in the general theory of engineering design.

Prior to this research, the theory of design in software engineering appropriate to support the reuse of software concepts had not been developed. This research represents a small step towards such a theory. Moreover, there was little reported work on the practical application of domain analysis to support reuse of software concepts

nor were there reports into the reuse of design structures. Through the development of the CDF and its application in the construction of design frameworks, this research has been able to demonstrate a more systematic approach to domain analysis and obtain results which were demonstrated to be potentially reusable. The work proceeded by studying related designs in specific domains using the CDF, and established its potential as a means of supporting the reuse of structural level design concepts. The large scale application of the CDF and demonstration of its support for reuse was carried out in collaboration with domain experts who were invaluable for validating the results of this work. The problem considered has been how to describe abstraction over structure in design. The research here addressed this problem from a very specific viewpoint of the architecture of process control systems. However, this thesis has shown that design frameworks are generally necessary to support the effective reuse of software concepts in an appropriate context during conceptual design. Through developing design frameworks based on the CDF, it has been shown how software concepts can be explicitly recorded within such frameworks and how these provide the basis for the reuse of software concepts in design.

References

[1] ABB. Deliverable F2.1: Reports on experiences with prototype, 12 April 1991. Practitioner Report P1094-ABB-0050.

[2] ACM. *Software Reusability, Volumes I and II.* ACM Press and Addison-Wesley, 1989.

[3] Martin Ader, Stephen McMahan, Gerhard Mueller, and Anna-Kristin Proefrock. The ITHACA technology a landscape for object-oriented application development. In *ESPRIT '90 Proceedings of the Annual ESPRIT Conference*, pages 31–51. Kluwer Academic Publishers, 1990.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[5] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

[6] Hanne Albrechtsen. Software concepts: Knowledge organisation and the human interface. In *Proceedings of the ISKO Conference.* Darmstadt, January 1990.

[7] Hanne Albrechtsen and Cornelia Boldyreff. Software classification, 15 Feb 1990. Practitioner Project Working Paper P1094-BrU-071.

[8] Hanne Albrechtsen and Lene Olsen. Methodology in programming linguistics within a real-time culture, 1 October 1990. Practitioner Project Report P1094-CRI-110/00.

[9] Christopher Alexander. *Notes on the Synthesis of Form.* Harvard University Press, 1964.

[10] Alvey. *Report on Software Engineering and Communications.* Department of Industry, U. K., June 1982. The report produced by the Alvey Committee.

[11] AMICE. *Open System Architecture for CIM.* Research Reports ESPRIT. Springer-Verlag, 1989. Written by the AMICE Consortium.

[12] Chris Anderson. Software reuse: A CAMP project update, 1988. Technical report from US Air Force Armament Laboratory, Elgin AFB, Florida 32542-5434, USA.

[13] J. R. Anderson. *Cognitive Psychology and Its Implications.* W. H. Freeman and Co., New York, 1985.

[14] Anon. Some research directions for large-scale software development. In *AT&T Technical Journal*, July/August 1988.

[15] H. Jack Barnard, Robert F. Metz, and Arthur L. Price. A recommended practice for describing software designs: IEEE standards project 1016. *IEEE Transactions on Software Engineering*, SE-12(2), February 1986.

[16] Victor R. Basili, Gianluigi Caldiera, and Giovanni Cantone. A reference architecture for the component factory. *ACM Transactions on Software Engineering and Methodolody*, 1(1):53–80, January 1992.

[17] LA Belady and MM Lehman. The characteristics of large systems. In MM Lehman and LA Belady, editors, *Program Evolution Processes of Software Change.* Academic Press, 1985.

[18] T. J. Biggerstaff and A. J. Perlis. Foreword on reusability. *IEEE Transactions on Software Engineering*, SE-10(5):474–476, September 1984.

[19] Ted J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22(7):36–49, July 1989.

[20] Ted J. Biggerstaff and Charles Richter. Reusability framework, assessment, and directions. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability*, volume I. ACM Press and Addison-Wesley, 1989.

[21] C. Boldyreff. Design methods for integrating system components. In P. A. V. Hall, editor, *Software Reuse and Reverse Engineering in Practice*, pages 81–97. Chapman & Hall, 1992.

[22] C. Boldyreff, P. Elzer, P. Hall, U. Kaaber, J. Keilmann, and J. Witt. PRACTITIONER: Pragmatic support for the reuse of concepts in existing software. In *Proceedings of Software Engineering 1990, Brighton, UK.* Cambridge University Press, 1990.

[23] C. Boldyreff and J. Zhang. From recursion extraction to automated commenting. In P. A. V. Hall, editor, *Software Reuse and Reverse Engineering in Practice*, pages 253–270. Chapman & Hall, 1992.

[24] Cornelia Boldyreff. Descriptive methods survey and guidelines for usage and recommendations on form and contents of external descriptions, 2 September 1988. Practitioner Working Paper P1094-BrU-Report-0009. A revised version of deliverable A1.1 combined with deliverable B1.1.

[25] Cornelia Boldyreff. Investigations concerning the representation of software concepts, February 1989. Practitioner Project Working Paper P1094-BU-CB-WPB2-WORKING PAPER-0022.

[26] Cornelia Boldyreff. The questionnaire: a generic form for the description of software concepts, 16 June 1989. Practitioner Working Paper P1094-BrU-0056.

[27] Cornelia Boldyreff. The questionnaire: a generic form for the description of software concepts, 16 June 1989. Practitioner Working Paper P1094-BrU-0056.

[28] Cornelia Boldyreff. Reuse, software concepts, descriptive methods and the practitioner project. *ACM SIGSOFT Software Engineering Notes*, 14(2), April 1989.

[29] Cornelia Boldyreff. Demonstration of the PRESS, 18 October 1990. Practitioner Working Paper P1094-BrU-104.

[30] Cornelia Boldyreff. Supporting system design from reusable design frameworks. In *Proceedings of the Second International Conference on INFORMATION SYSTEM DEVELOPERS WORKBENCH Methodologies, Techniques, Tools and Procedures, Gdansk, 25-28 September 1990.* University of Gdansk, 1990.

[31] Cornelia Boldyreff. Deliverable E4.2: Handbook for design of concept and module surfaces, 18 January 1991. Practitioner Report P1094-BrU-0109/01.

[32] Cornelia Boldyreff. A design framework for software concepts in the domain of steel production. In *Proceedings of the Third International Conference on INFORMATION SYSTEM DEVELOPERS WORKBENCH Methodologies, Techniques, Tools and Procedures, Gdansk, 22-24 September 1992.* University of Gdansk, 1992.

[33] Cornelia Boldyreff, Patrick Hall, and Jian Zhang. Reusability: The practitioner approach. In *Proceedings Workshop "Reuse" RESEARCH IN PROGRESS*, pages 1–7. Delft University of Technology, November 1989. Position paper presented at workshop.

[34] Cornelia Boldyreff and Uwe Krohn. The practitioner reuse support system (PRESS): A consideration from the standpoint of tool in terconnection. In *Annual Review of Automatic Programming*, volume 16, Part 2. Pergamon Press, 1992. Proceedings of the Fourth IFAC/IFIP Workshop on Experience with the Management of Software Projects, Austria, May 18-19, 1992.

[35] Richard Bornat. *Programming from First Principles.* Prentice-Hall International Series in Computer Science. Prentice-Hall, 1987.

[36] Frank Bott, Bob Gautier, and Mark Ratcliffe. CDL and its tools. In Frank Bott, editor, *ECLIPSE An integrated project support environment*, pages 181–188. Peter Pergrinus, Ltd., London, 1989.

37] R. J. Brachman. *A Structural Paradigm for Representing Knowledge.* Bolt, Beranek & Newman, 1978. Technical Report 3605.

38] J. M. Brady. *The Theory of Computer Science A Programming Approach.* Chapman and Hall, London, 1977.

39] David C Brown and B Chandrasekaran. *Design Problem Solving, Knowledge Structures and Control Strategies.* Research Notes in Artificial Intelligence. Pitman Publishing, London, 1989.

40] G F Bryant, W J Edwards, and C H McClure. Cold-rolling-mill control-system design, part 1: System description and control objectives. In G F Bryant, editor, *Automation of tandem mills.* The Iron and Steel Institute, London, 1973.

1] Louis L. Bucciarelli. An ethnographic perspective on engineering design. *Design Studies*, 9(3), July 1988.

2] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *Proceedings of Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Carnegie-Mellon University, Pittsburgh, 1977.

3] R. M. Burstall and J. A. Goguen. An informal introduction to specifications using CLEAR. In RS Boyer and JS Moore, editors, *The correctness problem in computer science.* Academic Press, 1981.

4] Richard L Campbell. An architecture for factory control automation. *AT&T Technical Journal*, 66(5):77-85, September/October 1987.

5] Lionel Cartwright et al. AISE software portability project — review of step 1. In *AISE Year Book*, 1984.

6] Lionel Cartwright et al. AISE software portability project — review of step 2. In *AISE Year Book*, pages 94–97, 1985.

7] Lionel Cartwright et al. AISE software portability project — review of step 3. In *AISE Year Book*, 1987.

[48] Peter Checkland. *Systems Thinking, Systems Practice.* Wiley, 1981.

[49] A. B. Chelyustkin. *The Application of Computing technique to Automatic Control Systems in Metallurgical Plant.* Metallurgizdat, Moscow, English edition, 1960. Translated by D. P. Barrett and published by Pergamon Press, 1964.

[50] B Cohen and M Jackson. A critical appraisal of formal software development theories, methods and tools, 1983. Technical Report, Standard Telecommunications Laboratories, Harlow, England, (ESPRIT Preparatory Study).

[51] CSTB. Scaling up: A research agenda for software engineering. *Communications of the ACM*, 33(3):281–293, March 1990. Excerpts from the report by the Computer Science and Technology Board.

[52] Bill Curtis, Herb Krasner, and Neil Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, November 1988.

[53] Tim Denvir. *Introduction to Discrete Mathematics for Software Engineering.* Macmillan Computer Science Series. Macmillan, 1986.

[54] F DeRemer and H Kron. Programming-in-the-large versus programming-in-the-small. In *IEEE Transactions on Software Engineering*, June 1976.

[55] E. W. Dijkstra. Introduction: Why correctness must be a mathematical concern. In RS Boyer and JS Moore, editors, *The Correctness Problem in Computer Science.* Academic Press, 1981.

[56] Liesbeth Dusink and Patrick Hall (Editors). *Software Re-use, Utrecht 1989.* Workshops in Computing. Springer-Verlag, 1991. Proceedings of the Software Re-use Workshop, 23-24 November 1989, Utrecht, The Netherlands.

[57] W J Edwards, J D Higham, C H McClure, and B J Mercer. Chapter 18 modelling programs for the design of cold-rolling-mill control systems. In G F Bryant, editor, *Automation of tandem mills.* The Iron and Steel Institute, London, 1973.

[58] J. Eekels and N. F. M. Roozenburg. A methodological comparison of the structures of scientific research and engineering design: their similarities and differences. *DESIGN STUDIES*, 12(4):197–203, October 1991.

[59] P. Elzer, Bent S. Jensen, Ulla Kaaber, Johannes Keilmann, Soren P. Nortoft, and J. Witt. Recommendations on the use of descriptive methods, 30 April 1987. Practitioner Working Deliverable A1.2.

[60] Peter Elzer, Rachel Jones, and Jan Witt. Practitioner — realistic reuse of software. In *Proceedings of GI Conference*, pages 507–515. Munich, October 1989.

[61] J Estublier, S Ghoul, and S Krakowiak. Preliminary experience with a configuration control system for modular programs. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* April 1984.

[62] S. I. Feldman. Make — a program for maintaining computer programs. *Software — Practice and Experience*, 9(4), April 1979.

[63] Christiane Floyd. Outline of a paradigm change in software engineering. *ACM SIGSOFT Software Engineering Notes*, 13(2), April 1988.

[64] Peter Freeman. Reusable software engineering: Concepts and research directions. In *Proceedings of the ITT Workshop on Reusability in Programming, Stratford, Connecticut.* ITT, Newport, RI, September 7-9, 1983.

[65] Peter Freeman. A conceptual analysis of the draco approach to constructing software systems. In P. Freeman, editor, *Tutorial: Software Reusability.* Institute of Electrical and Electronic Engineers, Washington. DC, 1987.

[66] G. Frege. Die grundlagen der arithmetik, 1884. Translated into English by JL Austin as The Foundations of Arithmetic, Revised , Basil Blackwell, Oxford (1968).

[67] John C. Funk and Larry McAllister. Controlling processes with DCS. In *Chemical Engineering*, pages 90–96, May 1989.

[68] Gandalf. The gandalf project. *Journal of Systems and Software*, 5(2), May 1985.

[69] J. A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16–28, February 1986.

[70] Siegfried Greif. The role of German work psychology in the design of artifacts. In John M. Carroll, editor, *Designing Interaction: Psychology of the Human-Computer Interface*, Cambridge Series on Human-Computer Interaction, pages 203–226. Cambridge University Press, 1991.

[71] Raymonde Guindon. Knowledge exploited by experts during software system design. *International Journal of Man-Machine Studies*, 33:279–304, 1990.

[72] Patrick Hall and Cornelia Boldyreff. Software reuse. In John A. McDermid, editor, *Software Engineer's Reference Book*. Butterworths, June 1990.

[73] Patrick A.V. Hall. A METAMODEL for software components and reuse, 1989. P1094-BrU-PH-Working Paper 0052.

[74] J. Heidepriem. Trends in process control of metal rolling. In *Proceedings of the 11th IFAC World Congress*, volume 11, pages 138–147. Tallinn/Estland, January 1990.

[75] P. Henderson and B. Warboys. An architectural framework for systems. In *ICL Journal*, May 1989.

[76] Peter Henderson and Brian Warboys. Configuration description for component reuse. In *Proceedings of the First International Workshop on Software Reusability, Dortmund, Germany, July 3-5, 1991*, pages 142–147. University of Dortmund, June 1991.

[77] CAR Hoare. An overview of some formal methods for program design. In *COMPUTER*, volume 20, pages 85–91. IEEE Computer Society, September 1987.

[78] Susan Hockey and Jeremy Martin. *Oxford Concordance Package Users' Manual*. Oxford University Computing Service, Oxford, U.K., 1988.

[79] J. W. Hopper and R. O. Chester. *Software Reuse Guidelines and Methods.* Plenum Press, New York, 1991.

[80] IEEE. *IEEE Recommended Practice for Software Design Descriptions.* The Institute of Electrical and Electronic Engineers, Inc., 1987.

[81] IEEE. *IEEE Tutorial: Software Reusability.* Institute of Electrical and Electronic Engineers, Washington, DC, 1987.

[82] IEEE. *IEEE Tutorial: Software Reuse: Emerging Technology.* IEEE Computer Society Press, Washington, DC, 1988.

[83] ISO. Documentation — guidelines for establishment and development of monolingual thesauri, 1986.

[84] ISO. Documentation — commands for interactive text searching (CCL), 1989.

[85] J. Christopher Jones. *DESIGN METHODS seeds of human futures.* John Wiley & Sons, this edition, 1980. contains a review of new topics.

[86] T. C. Jones. Reusability in programming: A survey of the state of the art. *IEEE Transactions on Software Engineering*, SE-10(5), September 1984.

[87] K. Kandt. Pegasus: A tool for the acquisition and reuse of software designs. In *Proceedings of COMPSAC 84*, pages 288–293. IEEE Computer Society Press, Silver Spring, MD, USA, November 1984.

[88] J. Keilmann and P. Elzer. Experience with the manual identification and selection of program concepts and modules, 10 February 1989. Practitioner Working Paper P1094-ABB-KE-WPC2.1-Report-0019.

[89] Johannes Keilmann. Order processing in a strip processing line, (a set of questionnaires completed at ABB' s industrial plants process control division), 1988.

[90] Johannes Keilmann. Faceted thesaurus, 20 February 1990. Practitioner Working Paper P1094-ABB-0032.

[91] R. J. Kermitz, F. J. Bold, and H. Bydalek. Reference-plant design adapts to restrictive site requirements. In *Power*, pages S–47 — S–52, April 1989.

[92] DE Knuth. Literate programming. *The Computer Journal*, 27(2), May 1984.

[93] Charles W. Krueger. *Models of Reuse in Software Engineering.* Carnegie Mellon University, 14 December 1989.

[94] Lancaster. The dragon reuse toolset. In *Proceedings Workshop "Reuse" RESEARCH IN PROGRESS*, pages 1–5. Delft University of Technology, November 1989. Position paper presented at workshop by N. Hadley from the Dragon Project at Lancaster.

[95] M. M. Lehman, V. Stenning, and W. Turski. Another look at software design methodology. *ACM SIGSOFT Software Engineering Notes*, 9(2), April 1984.

[96] M. D. Lubars and M. T. Harandi. Knowledge-based software design using design schemas. In *Proceedings of the 9th International Conference on Software Engineering*, pages 253–262. 30th March — 2nd April 1987.

[97] Mitchell D. Lubars. AFFORDING HIGHER RELIABILITY THROUGH SOFTWARE REUSABILITY. *ACM SIGSOFT Software Engineering Notes*, 11(5):39–42, October 1986.

[98] Allen Macro and John Buxton. *The Craft of Software Engineering.* International Computer Science Series. Addison-Wesley Publishing Company, 1987.

[99] John A. McDermid. Introduction and overview to Part II, Methods, Techniques and Technology. In John A. McDermid, editor, *Software Engineer's Reference Book.* Butterworths, June 1990.

[100] Mihajlo D Mesarovic. Multilevel systems and concepts in process control. *Proceedings of the IEEE*, 58(1):111–125, January 1970.

[101] MIT-JSME. Computer-aided cooperative product development, 1991. Proceedings of the MIT-JSME Workshop, MIT, Cambridge, USA, November 1989.

[102] MWG. *Handbook of the European Iron and Steel Works.* Montan- und Wirtschaftsverlag GmbH, Frankfurt am Main, 1985. 8. Auflage, printed in German, English and French.

[103] J. Neighbors. The draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, SE-10(5):564–573, September 1984.

[104] Sandra L. Newsome and William R. Spillers. Tools for expert designers: Supporting conceptual design. In *Design Theory '88*, pages 49–55. Springer-Verlag, 1989.

[105] Christopher Norris. *DERRIDA.* Fontana Modern Masters. Fontana Press, London, 1987.

[106] NSF. *Design Theory '88.* Springer-Verlag, 1989. Proceedings of the 1988 NSF Grantee Workshop on Design Theory and Methodology.

[107] Anthony A. Oettinger. Letter to the ACM membership. *Communications of the ACM*, 9(8), January 1966.

[108] Lene Olsen and Kim Bisgaard. PRESS design description, 18 December 1989. Practitioner Project Report P1094-CRI-LO+KIB-WPE2.2-Report-0086/01.

[109] Lene Olsen et al. First studies on programming linguistics within a real time culture, January 1989. Practitioner Project Report P1094-CRI-HAL-WPC4.3-0068.

[110] Gerhard Pahl and Wolfgang Beitz. *Engineering Design a systematic approach.* Springer-Verlag, English edition, 1988. This published 1988 in the United Kingdom by The Design Council has been translated by Arnold Pomerans and Ken Wallace. The title of the original in German is Konstruktionslehre: Handbuch fuer Studium und Praxis, and it was first published in 1977.

[111] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 5(12):1053–1058, December 1972.

[112] David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2), February 1986.

[113] D E Perry and W M Evangelist. An empirical study of software interface faults — an update. In *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences*, volume II, pages 113–126. January 1987.

[114] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. In *ACM SIGSOFT Software Engineering Notes*, volume 17, pages 40–52. ACM, 1992.

[115] Henry Petroski. *To Engineer Is Human The Role of Failure in Successful Design.* Macmillian, London, 1985. (First published in the U. S. A. in 1982.).

[116] Henry Petroski. *The Pencil A History of Design and Circumstance.* Faber and Faber, 1989.

[117] Maciej Pietrzyk, Jan Kusiak, and Miroslaw Glowacki. Some aspects of development of models for automatic control of rolling mills. *steel research*, 61(8):359–364, 1990.

[118] George Polya. *How to Solve It.* Princeton University Press, second edition, 1957.

[119] Practitioner. ESPRIT project P1094 — PRACTITIONER: A support system for pragmatic reuse of software concepts, technical annex, version 3, 25 May 1987. Written by members of the Practitioner Consortium.

[120] Richard Preston. Annuals of enterprise, Hot Metals-I. In *New Yorker*, pages 43–71, 25 February 1991.

[121] Richard Preston. Annuals of enterprise, Hot Metals-II. In *New Yorker*, pages 41–79, 4 March 1991.

[122] R. Prieto-Diaz. Domain analysis for reusability. In *COMPSAC87 Conference, Tokyo, Japan, October 7-9.* 1987.

[123] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, January 1987.

[124] R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4):307–334, November 1986.

[125] Ruben Prieto-Diaz. Classification of reusable modules. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability*, volume I. ACM Press and Addison-Wesley, 1989.

[126] Ruben Prieto-Diaz. Domain analysis: An introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47–54, April 1990.

[127] Ruben Prieto-Diaz. Implementing faceted classification for software reuse. In *Proceedings of the 12th International Conference on Software Engineering*, pages 300–304. IEEE Computer Society Press, 1990.

[128] Ruben Prieto-Diaz and Guillermo Arango. *Domain Analysis and Software System Modelling.* IEEE Computer Society Press Tutorial. IEEE Computer Society Press, 1991.

[129] Ruben Prieto-Diaz and Guillermo Arango. *Domain Analysis and Software System Modelling.* IEEE Computer Society Press Tutorial. IEEE Computer Society Press, 1991.

[130] Michael Rabins, David Ardayfio, Steven Fenves, Ali Seireg, Herbert Richardson, and Howard Clark. DESIGN THEORY AND METHODOLOGY — a NEW DISCIPLINE. In *MECHANICAL ENGINEERING*, pages 23–27, January 1986.

[131] Brian Randell. System design and structuring. *The Computer Journal*, 29(4), 1986.

[132] M. Ratcliffe. Report on a workshop on software reuse held at hereford, UK on 1,2 may 1986. *Software Engineering Notes*, 12(1):42–47, January 1987.

[133] Howard B Reubenstein and Richard C Waters. The requirements apprentice: Automated assistance for requirements acquisition. *IEEE Transactions on Software Engineering*, 17(3):226–240, March 1991.

[134] C Rich. A formal representation for plans in the programmer's apprentice. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence.* Vancouver, January 1981.

[135] C Rich and H Shrobe. *Initial report on a LISP programmer's apprentice.* Massachusetts Institute of Technology, December 1976.

[136] C Rich, HE Shrobe, and RC Waters. An overview of the programmer's apprentice. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence.* Tokyo, January 1979.

[137] Charles Rich and Richard C. Waters. The programmer's apprentice. In *COMPUTER*, volume 21. IEEE Computer Society, November 1988.

[138] Donald A. Schon and Louis L. Bucciarelli. Design theory and methods – an interdisciplinary approach. In Sandra L. Newsome, W. R. Spillers, and Susan Finger, editors, *Design Theory '88*, pages 29–35. Springer-Verlag, 1989.

[139] Ian Sedwell, Ulla Kaaber, and Hanne Albrechtsen. The linguistic analysis of the unix on-line documentation, November 1988. Practitioner Project Deliverable P1094-BU-IS-WPC4.1-REPORT-0021.

[140] Mary Shaw. Larger scale systems require higher-level abstractions. In *Proceedings of the Fifth International Workshop on Software Specification and Design.* IEEE Computer Society, 1989.

[141] Mary Shaw. Elements of a design language for software architecture, January 1990. Position Paper for IEEE Design Automation Workshop.

[142] Mary Shaw. *Informatics for a New Century: Computing Education for the 1990s and Beyond.* Carnegie-Mellon University. Software Engineering Institute, Pittsburgh, July 1990. Technical Report, CMU/SEI-90-TR-15, ESD-90-TR-216.

[143] E. H. Sibley. A layered approach to very large system specification. In Bruce D Shriver, editor, *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, volume II, Software Track, pages 988–995. IEEE Computer Society Press, 1989.

[144] Herbert A. Simon. *The Sciences of the Artificial.* The MIT Press, second edition, 1981.

[145] Harry M. Sneed and Gabor Jandrasics. Inverse transformation from code to specification. In *Proceedings Software Tools '89.* Blenheim Online, London, 1989.

[146] I Sommerville and R Thomson. The ECLIPSE system structure language. In *Proceedings of the 19th Annual Hawaii International Conference on System Sciences.* 1986.

[147] Ian Sommerville. *Software Engineering.* Addison-Wesley, fourth edition, 1992.

[148] Ian Sommerville, John Mariani, Neil Hadley, and Ronnie Thomson. Unpublished paper, Software Design with Reuse, obtained from Neil Hadley, 1989.

[149] Ian Sommerville and Murray Wood. A software components catalogue. In R. Davies, editor, *Intelligent Information Systems: Progress and Prospects*, pages 13–32. Wiley, 1987.

[150] T. A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, SE-10(5):494–497, September 1984.

[151] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, 1981.

[152] Andrew S. Tanenbaum. *Structured Computer Organisation.* Prentice/Hall International, Inc., second edition, 1984.

[153] Harold Thimbleby. Delaying commitment. In *IEEE Software*, May 1988.

[154] WF Tichy. *Software Development Control Based on Systems Structure Description.* Carnegie-Mellon University, Computer Science Department, January 1980. PhD Thesis.

[155] W. J. Tracz. *Formal specification of Parameterized Programs in LILEANNA.* Stanford University, 1990. PhD Dissertation Draft.

[156] Will Tracz. Ada reusability efforts: A survey of the state of practice. In *Proceedings of the Fifth Annual Joint Conference on Ada Technology and Ada Washington Symposi um*, pages 35–44. 1987. Reprinted in Tracz's IEEE Tutorial on Software Reuse.

[157] Will Tracz. The three cons of software reuse. In *Proceedings of Third Annual Workshop: Methods and Tools for Reuse.* CASE Centre, Syracuse University, NY, USA, June 13-15, 1990.

[158] Wladyslaw M. Turski and Thomas S.E. Maibaum. *The Specification of Computer Programs.* Addison-Wesley Publishing Company, 1987.

[159] UNICOM. *Software Reuse and Reverse Engineering in Practice.* Chapman & Hall, 1992. Collection of papers presented at two Unicom seminars held in London in 1989 and 1990, edited by P. A. V. Hall.

[160] USS/AISE. *The Making, Shaping and Treating of Steel.* Association of Iron and Steel Engineers (USA), tenth edition, 1985.

[161] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker (Editors). *Revised Report on the Algorithmic Language Algol 68.* Springer-Verlag, 1976.

[162] L. von Bertalanffy. The organism considered as a physical system, 1940. Reprinted in Bertalanffy, L. von, General Systems Theory, Braziller, New York, 1960.

[163] L. von Bertalanffy. General systems theory — a critical review. *General Systems*, VIII:1–20, 1962.

[164] Lev S Vygotsky. *Thought and Language.* Soc.-econom. izd., Moscow-Leningrad, 1934. Translated and Edited by E Hanfmann and G Vakar, published by MIT Press, 1962.

[165] Martin Ward. Transforming a program into a specification, January 1988. Computer Science Technical Report 88/1, School of Engineering and Applied Science, University of Durham.

[166] Richard C. Waters and Yang Meng Tan. Towards a design apprentice: Supporting reuse and evolution in software design. *ACM SIGSOFT Software Engineering Notes*, 16(2):33–44, April 1991.

[167] Herbert Weber. The integration of reusable software components. In *Journal of System Integration*, pages 55–79. Kluwer Academic Publishers, 1991.

[168] Peter Wegner. Varieties of reusability. In *Proceedings of the ITT Workshop on Reusability in Programming, 7-9 September 1983, Stratford, Connecticut*, pages 30–44. ITT, Newport, RI, 1983.

[169] Theodore J. Williams. *Tasks and Functional Specifications of the Steel Plant Hierarchical Control System.* Purdue Laboratory for Applied Industrial Control, School of Engineering, Purdue University, West Layfayette, Indiana 47907, USA, 1984. Volume 1, Chapters 1-15, Volume 2, Chapters 16-21, Report No. 98, Originally prepared Sept. 1977, Expanded June 1982, Revised and Further Expanded June 1984.

[170] Terry Winograd and Fernando Flores. *Understanding Computers and Cognition: A New Foundation for Design.* Ablex Publishing Corporation, New Jersey, 1986.

[171] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, September 1990.

[172] F. Wolff. Long-term controlling of software reuse. *Information and Software Technology*, 34(3):178–184, March 1992.

[173] J. C. P. Woodcock. Structuring specifications in Z. *Software Engineering Journal*, 4(1):55–66, January 1989.

[174] H. Zemanek. Formalization: Past, present and future. In B. Shaw, editor, *Formal Aspects of Computing Science.* Newcastle University, 1974. Reprinted in Lecture Notes on Computer Science 23: Programming Methodology, Springer (1975).

[175] Jian Zhang and Cornelia Boldyreff. Towards knowledge-based reverse engineering. In *Proceedings of the Fifth Annual Knowledge-Based Software Assistant Conference, 24-28 September 1990.* Syracuse, NY, 1990.

[176] H. Zimmermann. OSI reference model — the OSI model of architecture for open systems interconnection. *IEEE Transactions on Communications*, COM-28, April 1980.

Appendix A

An Overview of the Practitioner Project

The aim of this appendix is to give an account of the Practitioner Project setting the context within which this research was undertaken. It consists of a brief introduction to the project and an overview of the project's approach to modelling reuse. Modelling is covered by discussing three aspects of the project's work:

1. the questionnaire, a form developed to be used in obtaining descriptions of reusable software concepts,
2. the thesaurus to be used to record terminology of the application domain relevant to indexing the software concept descriptions, and
3. a metamodel of the reuse process.

The relationship established by the project between the questionnaire and thesaurus is clarified. The project developed a faceted thesaurus, i.e. a thesaurus in which the terms are listed under distinct classes known as facets. The role of facets in supporting domain views is discussed, and it is shown how facets within a view can

be used to systematically classify software concept descriptions. Refinement of the metamodel resulted in the formulation of methods to support both design-for-reuse and design-with-reuse. A short account of these is given and the role of design frameworks is discussed. This appendix concludes by identifying the main areas of research within the project to which the author contributed and by listing the author's publications resulting from work on the Practitioner project.

A.1 An Introduction to the Practitioner Project

The Practitioner Project was a collaboration between Asea Brown Boveri AG and PCS Computer Systeme GmbH in Germany, Computer Resources International in Denmark, and three universities: Brunel University, the Technical University Clausthal and the University of Liverpool. The project completed its work in November 1991. The ultimate goal of this five year project was the development of a support system for the pragmatic reuse of software concepts; and a major result of the project was the development and demonstration of the Practitioner REuse Support System (PRESS). The project was concerned with the reuse of software concepts from designs through to code, focusing on concepts realised in existing software rather than the formulation of practices to be applied in the development of new reusable software [60]. The motivation for investigating reuse of existing software was to allow the exploitation of the investments already made in its development and maintenance. The concentration on the reuse of designs at a conceptual level was in part because the main domain in which reuse was considered during the project was process control in Steel Production. In the field of process control software where use of assembly language is commonplace and code reuse is often not feasible, there is a high degree of commonality in design concepts employed, and hence the focus was on reuse at this level. Practitioner also aimed to support the designer early on in the design process, and hence its concentration was on software concept reuse in general.

Initially there were three major areas of concern within Practitioner:

- description of software concepts, selection of appropriate forms of description to support reuse, and assistance with description by analysis of existing software documentation and source code;
- classification, storage and retrieval of software concept descriptions to enable a software engineer to select the software design or code appropriate to meet specific requirements;
- development of methods and tools to support the software engineer in the construction of software systems from reusable software concepts.

The research supporting this thesis has contributed primarily to the first and third of these concerns. The notion of a software concept as a potentially reusable object comes from accounts of software engineers' design experience; often, an engineer is able to formulate a solution to a new system requirement by following up informal ideas based on an understanding of existing software. These reusable ideas are the software concepts of the individual practitioner, i.e. software engineer; they may be fundamental ideas from Computer Science such as *queue* or a *stack*, or operational ideas such as *order processing*, or very specific application ideas such as *material tracking in a strip processing line.*

Terminology analysis was the method used to establish the terms used in a particular domain to describe its software concepts. In Practitioner, the domain of Steel Production was studied in depth [22, 32]. A thesaurus was developed working from terminology sources in standardised form, e.g. existing thesauri, standards, dictionaries and glossaries, and sources in non-standardised form such as technical publications and project documentation [90].

In parallel with the terminology analysis, existing domain software was described using a questionnaire developed by the project [26] and described in more detail in Section 2.2.3. Although software concept descriptions are given a standardised structure via the questionnaire, use of a particular language or descriptive method to describe the concept is not dictated. It is assumed that the application oriented descriptions of concepts in the questionnaires will be in the domain language, typ-

ically a natural language. The form of the questionnaire allows complex software concepts to be described by a process of recursive decomposition into atomic software concepts. A large application concept is described by a set of questionnaires; for example, over thirty questionnaires were required to describe *Order processing in a Strip Processing Line* [88].

This work laid the ground for the project's approach to supporting software concept reuse; and it provided the initial population of the Practitioner REuse Support System described below.

Over the first two years of the project during Phase I of the research, the emphasis was on collection of software concepts in specific application domains and investigation into appropriate descriptive methods to support reuse. Various approaches to reuse support were investigated, and two pre-prototype reuse support systems were developed using UNIX and the PCTE.

Phase II was concerned with the development of a prototype Practitioner REuse Support System (PRESS). In developing a reuse support system, the project was determined to reuse existing software concepts. A standard language for interactive text searching, CCL [84], has been used in the PRESS to support the searching for software concepts. The ISO Guidelines for thesaurus construction [83] formed the starting point for the PRESS on-line thesaurus.

The links between the thesaurus and the database of software concepts are both static and dynamic. Terminology used in the questionnaires to describe concepts is *up-lifted* to the preferred terminology by making use of the thesaurus when concept descriptions are installed in the PRESS. During a search to find relevant reusable concepts, the PRESS makes active use of the thesaurus through an extension to CCL that was defined by the project: the thesaurus operator. The thesaurus operator enables search terms to be augmented by additional terms based on standard thesaurus relations, such as *broader terms*, *narrower terms*, etc.

The PRESS has been implemented using an existing database management system

for the software concept store and thesaurus. The user views these through browsers built from a standard windowing system. Figure A.1 - Data Flow Diagram of the PRESS - shows the main components of the PRESS and the dataflows between them.

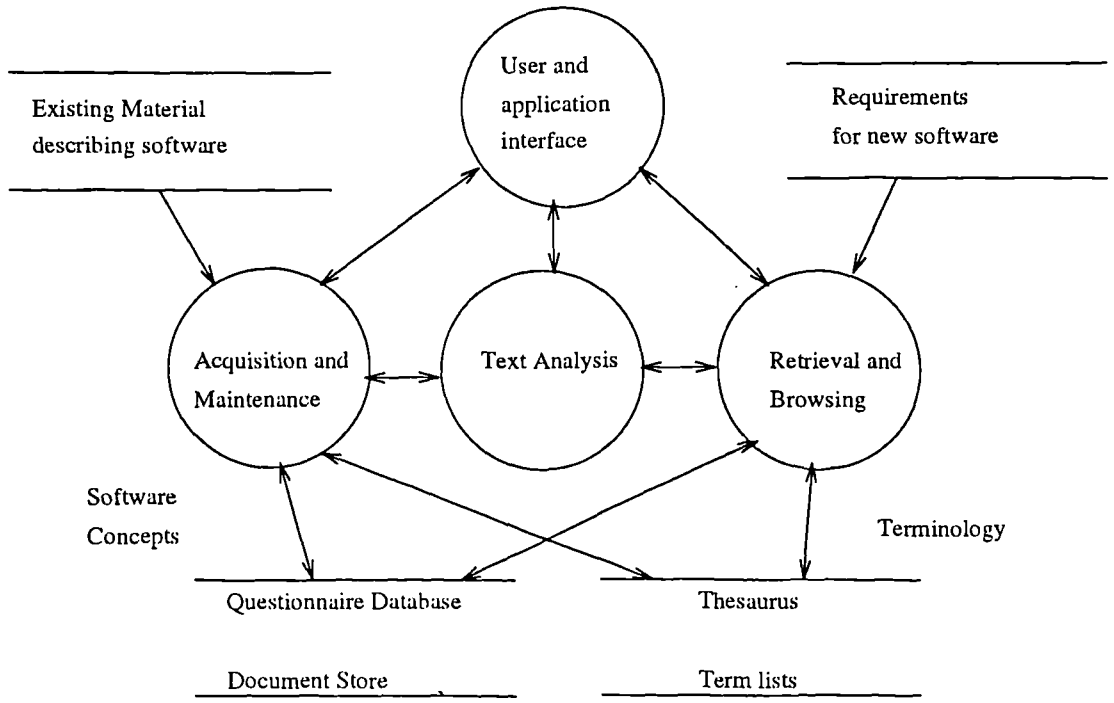


Figure A.1: Data Flow Diagram of the PRESS

The PRESS as developed has a low cost, easily installed form, PRESSTO, and a more sophisticated form, PRESSTIGE. The latter has been interfaced with other tools such as Cadre's Teamwork CASE tool and an experimental multi-user hypertext system, MUCH (Many Users Creating Hypertext), from the University of Liverpool.

This prototype was experimentally evaluated during Phase III. Demonstrating the payback on reuse in the application domain studied, process control systems in Steel Production, was rather difficult as software developments in steel mills are likely to

take place over a number of years. Demonstrating reuse in a realistic development would have required a timescale greater than the remaining lifetime of the project. However, within ABB, the process of *offer preparation* was identified as one where reuse of software concepts could have an immediate impact. Responding to calls for tender, by preparing *offers*, i.e. proposals to build the required software, is an area where reuse typically has not been considered; and yet it provides a short enough illustration of reuse without being too simplistic for demonstration purposes while at the same time bearing enough resemblance to the design process as a whole to make realistic use of the Practitioner methods and tools. It was also possible to build on informal reuse of material in offer preparation already practiced by experienced engineers in the metallurgy division of ABB. In conjunction with experimental use of the PRESS to support preparation of offers, aspects of modelling the economics of reuse were studied [172]. In the final Phase IV of the project, the prototype PRESS was refined in light of the experience gained and in light of various research studies.

The academic organisations in the consortium were not directly involved in the prototype development or evaluation; their contribution to the project throughout was directed towards the more fundamental research areas of the project described below in greater detail. Addressing the methodological aspects of populating the PRESS and employing it in the design of new systems - design-for-reuse and design-with-reuse - were their main concerns. Before discussing these in detail, an overview of the project's modelling of reuse will be given together with an introduction to the canonical form developed by the project for the description of software concepts - the questionnaire.

A.2 Overview of Practitioner Project's Modelling of Reuse

In modelling reuse within the Practitioner Project, two primary aspects of reuse were considered:

- *the objects of reuse* i.e. What is reusable from one software project to the next? and
  How can such reusable elements be best described to facilitate their reuse?
- *the process of reuse* i.e. What constitutes reuse in Software Engineering? and
  How is reuse best practised and supported?

Within Practitioner the focus was on supporting the reuse of software concepts for the reasons already discussed above. The project used the following working definition of a software concept:

> *an abstract task, described by its purpose (and/or goal), the related objects and/or functional principles of the underlying mechanism (which will be typically, but not necessarily, of an algorithmic nature).*

[119]

At the highest level of description, a task is equated with an application. A software concept may be much less complex than an application; for example, the software concept, calculation of rolling forces, is an abstract task realised in software in the Steel Production domain.

The reuse process on which the Practitioner Project focussed encompasses the identification of reusable software concepts as embodied in existing software systems. A distinction is made between software concepts which are identifiable as parts of a whole system, that is software components; and common application frameworks or functional structures, that is system architectures [28]. Both are characterised as software concepts. In both cases, a further distinction is made between specific instances and generic forms of both component concepts and architectural concepts.

The questionnaire, i.e. a descriptive form for software concepts, was developed to give a uniform structure to all software concept descriptions. The development of this form and its adequacy is considered in depth in Chapter 3. Understanding the

role of the questionnaire is critical to an understanding of the work done within the project, as it provided the basis for both the identification of reusable objects and their classification. Not only did the questionnaire give an external form to all descriptions; a development of this form was reflected by a common internal representation of all reusable software concepts held in the PRESS, and this common representation provided a basis for software composition and tool integration within the PRESS.

A programme of research on *program linguistics* was developed which applied the techniques and tools of computational linguistics to application oriented descriptions obtained via the questionnaire and from existing software documentation in order to assist with terminology analysis and thesaurus construction [139]. Furthermore, to automate the building up of the software concept database, the reverse engineering of designs from source code was investigated [23].

A model of the reuse process was developed which encompassed population of the software concept database by a process of abstracting concepts from studies of existing software systems, their generic description and classification. Software concepts may be retrieved from the PRESS database by a process of guided selection using the thesaurus. To reuse fully a retrieved software concept, the designer may need to specialise or modify and adapt it in some way and finally compose it with other concepts to form a new application. For this reason, in the research, a need to consider provision of assistance with design with reusable concepts was identified.

A.2.1 The Questionnaire - Software Concept Descriptive Form

The original questionnaire was devised early on in the first year of the project with the idea that a common framework was required for concept descriptions [59]. Concepts were described from three viewpoints: the microworld of the concept's application domain; the functional and structural composition of the concept; and

the macroworld of the concept's historical development. The original questionnaire consisted of three main parts corresponding to these viewpoints:

Part A - Application Oriented Description
Part B - Implementation Oriented Description
Part C - Historical Development

Software concepts are described with the questionnaire via a process of recursive decomposition. At the top level, a concept's immediate parts are identified. If these are atomic concepts, they are described directly; otherwise a new questionnaire is required for each non-atomic subconcept [33].

Although the questionnaire was developed to reflect good practice, it was gratifying to note that it followed the IEEE Recommended Practice for Describing Software Designs [15, 80]; an analysis of the similarities can be found in [24]. The suitability of this form for representing both horizontal and vertical composition and the possibility of transforming questionnaire descriptions into other representational forms is discussed in [25]. From its inception the questionnaire was practically employed within the project and in the light of this experience and studies of other representations to support reuse, the original questionnaire form was developed to reflect a better understanding of the essential features of concept description [27]. These developments will form the main topic of the following chapter.

The problem of providing guidance on how to decompose a concept into its constituent parts did not go away with the new form of the questionnaire. It was concluded that decomposition must be recognised as requiring expert knowledge about the concept and expert judgement. It was found to be equally difficult to choose the appropriate levels of abstraction needed to describe software concepts needed to cover the concept derivation. The approach of recording software concepts working from general system models described in chapters 4 and 5 has been developed in part to address these problems. This has led onto the development of design frameworks described in chapters 5 and 6.

Within the project, experience with the questionnaire was varied. It was found that where a department employed a high standard of software documentation during its development, it was possible partially to automate extraction of relevant information required to fill in the questionnaire. In other cases where the software was not so well documented, filling in the questionnaire has proved difficult as well as tedious, and a clear need for tools to support such reverse engineering was recognised. Efforts to address this need are described in the following section.

Although the questionnaire was developed as a tool for gathering information about software concepts for populating the PRESS, it was recognised to have other uses within the Practitioner project. The questionnaire also provided a standard format for describing new software constructed from reusable concepts allowing new applications developed with the PRESS to be recycled through the support system and made available themselves for reuse.

Reverse Engineering of Software Descriptions

The Practitioner project sought to apply the methods of Reverse Engineering to software in order to gain a better understanding of its underlying concepts and subsequently to support their reuse [23]. Searching for a satisfactory solution, sometimes without any reassurance that a solution is possible, presents a challenge in engineering a new system especially at the start of development where lacking a conceptual unity to the form and content of the new system, the designer may take various wrong turnings. Throughout the development process, the designer continues to prune away various possible solutions until the end-point of the development is reached. This can be contrasted with the process of examining an existing software system and recasting what has been developed into a more understandable form. This process is called reverse engineering. When reverse engineering a software system, an attempt is made, not to recreate the history of its development, but rather, through examining existing software, to establish its underlying theory and concepts. The principal artifact that is usually available is the source code of the system; thus, a major task of reverse engineering is to develop or extract a higher level description of the software from its lower level code description [137, 145]. Typ-

ical tasks of reverse engineering include design recovery [19] and code transformation to obtain a specification [165].

Reverse engineering was relevant to the following goals of the Practitioner project:

- identify reusable components from existing code;
- recover application oriented descriptions of reusable components if they do not previously exist;
- abstract specifications of reusable components;
- classify or generalize identified reusable components as software concepts;
- support automated filling in of questionnaires for software concepts;

As a case study towards reverse engineering of software concept descriptions in the Practitioner project, a code transformation system was developed to transform program constructs into higher level recursive forms, for details, see [23]. The system is aimed at producing automated semi-natural language documentation of software with user intervention. As a result of this work, the need for a more expansive view of reverse engineering was recognised [175].

A.2.2 The Thesaurus

In order to reuse software within the PRESS, not only must the software be adequately described, it must also be possible for the collection of software concepts itself to be comprehensible and its contents made accessible to its potential users. Indexing of concepts based on development of a faceted thesaurus described below provided a solution to the accessibility problem. For dealing with overall comprehensibility of the collection, i.e. of describing concept classes of the concept database, within Practitioner, the method of faceted classification was chosen. Faceted classification is a flexible method of classification suitable for describing dynamic collections

of concepts; its application to software has been demonstrated by Prieto-Diaz and Freeman [123]. For a review of approaches to software classification undertaken within the frame of the project, see [7]. Below the way in which a faceted thesaurus can be used to derive a systematic classification of software concepts is described; such a classification is known as a faceted classification because the classification headings have been synthesized from terms under each facet of the thesaurus.

For the purpose of indexing and classification, the natural language text of the questionnaire and associated documentation from the application domain were analysed using conventional text analysis techniques and tools, such as the Oxford Concordance Package (OCP) [78]. From the concordances and collocations obtained, terminology for classification was obtained in a semi-automated process although in the final instance it was necessary to consult with a domain expert. The individual terms may be related by means of a thesaurus. In the case of Practitioner where both English and German versions of questionnaires were analysed, the initial emphasis was on the development of a mono-lingual thesaurus where German translations were simply added to each term entry. English was the primary technical language on the project. In the work on thesaurus construction, the project elected to follow the ISO standard for this purpose [83] although it was necessary to extend this standard to cover the construction of an on-line thesauri.

Thesaurus entries consist of an individual term with all its known relations as defined in the ISO standard and some additions described below. The relations described in the standard are as follows:

top terms,

broader terms,

narrower terms,

used for terms, and

related terms

All of these relations are given for a *preferred term.* In the case of non-preferred

terms, only the *used for terms* are given. Relations for listing *new terms* and *old terms* for each entry were added to enable a record of the history of changes in terminology to be made. The attributes to a *preferred term* described in the standard are:

scope note including definition,

date indexed.

There are no attributes described for a *non-preferred term.* A separate definition and a translation (usually in German) have been added to the standard entry. The definition was used by the project to record the term's facet and its associated view. To assist with thesaurus maintenance and general exploration of terminology, the PRESS development included a thesaurus browser. This allows the user to view a thesaurus entry through a set of windows as shown in Figure A.2 - Typical Thesaurus Entry as Viewed Using the PRESS Browser. This figure is reproduced from [29].

In Figure A.2, in the top line of the main window, a request for the thesaurus entry for the term, *finish line*, has been made. As this is a preferred term, it has a full set of relations recorded in the thesaurus which are displayed below in the smaller windows. Reading these from the centre, it can be seen that this term, *finish line*, was entered on the 7th June 1990 and its translation in German is *Adjustagelinien.* Moving now in clockwise order starting with the window immediately above the centre, the broader term (BT) is *process line*; the definition indicates that *finish line* falls under the facet, *Equipment (by processes)*; there is no scope note; two related terms are: *cold roll mill (plants)* and *hot roll mill (plants)*; there is no known old term; a long list narrower terms (NT) terms is given (these are kinds of finishing lines); there is no known new term; the term *finish plant* is used for (UF) *finish line* i.e. *finish plant* is the non-preferred term; and finally the top term (TT) for *finish line* is *process line.*

As a result of work organising the PRESS contents, the importance of domain analysis to support indexing and classification of software was recognised as well as the need for guidance from a domain expert to ensure usability of the PRESS within

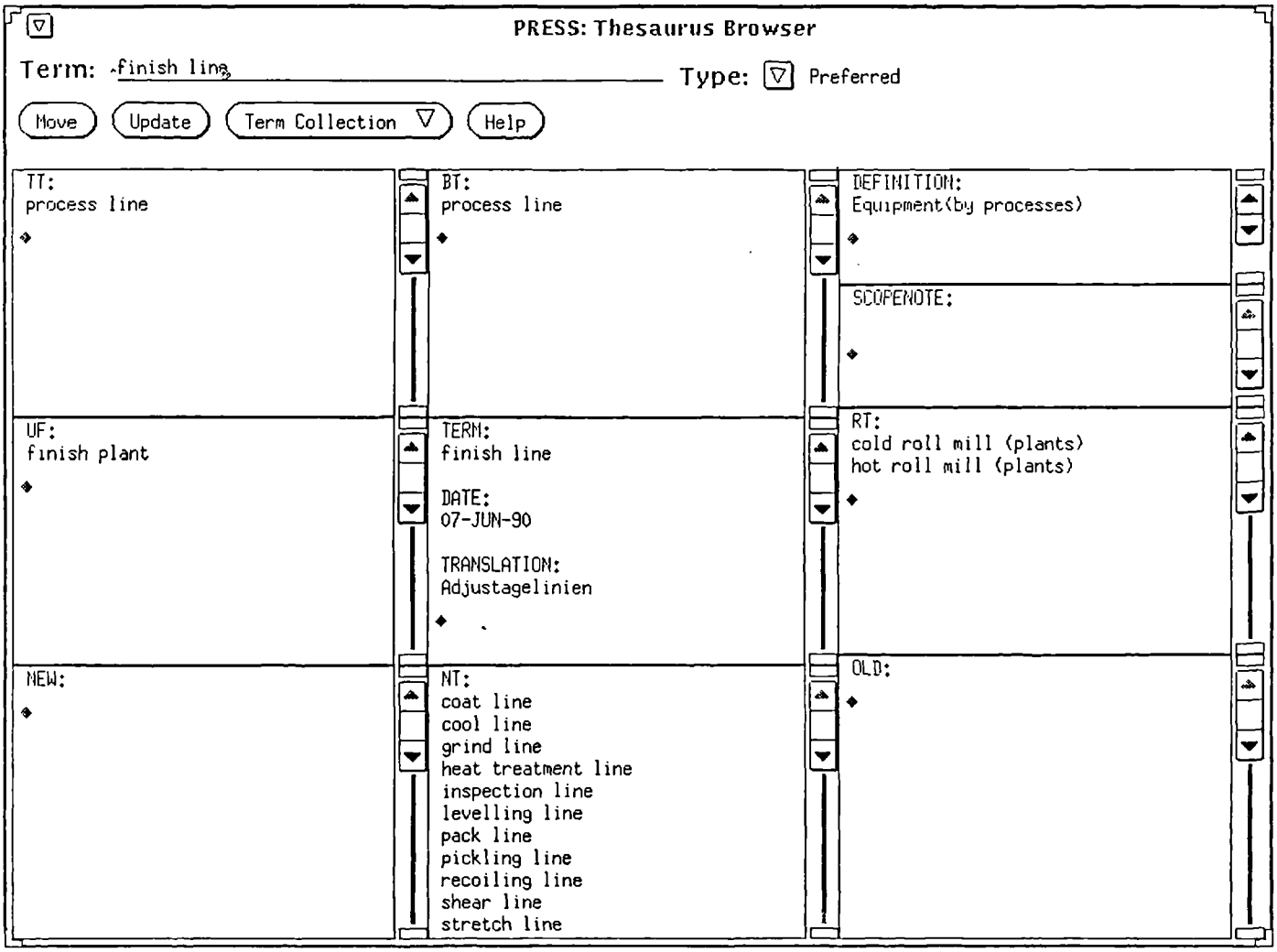


Figure A.2: Typical Thesaurus Entry as Viewed Using the PRESS Browser

a particular domain. Here the project's conclusions were very similar to those of the Draco Project [103, 65].

The Relationship between the Questionnaire and the Thesaurus

It was important within the Practitioner project to clarify the relationship between the terminology used in the domain, which in Practitioner, was recorded in the form of a thesaurus, and the software concepts identified and recorded on questionnaires (and later using the Concept Desrciption Form developed from the questionnaire). The latter were largely obtained as a result of field studies of existing software from a particular domain. Since the sources proposed for thesaurus construction could also form material for questionnaire input, the work on thesaurus construction and maintenance proceeded in parallel with software concept capture, i.e. questionnaire filling. In so far as terms used in the domain correspond to to domain concepts, there is some overlap between the thesaurus terms and the terms used to identify and describe the software concepts of that domain. One could also reasonably expect that there will be similar relationships between terms in the thesaurus and concepts described by questionnaires; this depends on a common granularity of terms and concept names resulting from the domain analysis. More important is the coverage of the thesaurus for the terminology used in the questionnaire descriptions of concepts; this can be gauged by comparing the actual terms used in the searchable fields of the concrete questionnaires and those recorded in the thesaurus. Such determination of coverage could perhaps provide some basis for maintaining the thesaurus as the questionnaire base is expanded, so that the thesaurus effectively supports the designer searching through the questionnaire base.

In what follows, the relation between the thesaurus and the questionnaire is examined only in so far as it relates to the use of the PRESS in design. The designer using the PRESS requires guidance to formulate successful searches for reusable concepts. In particular, guidance is needed to determine how domain terminology used to express the initial requirements and design issues is related to descriptions of existing software concepts in the domain. The thesaurus provides the required

link.

The links currently implemented between the thesaurus and the database of software concepts can be static or dynamic. Terminology used in the questionnaire to describe concepts is *up-lifted* to the preferred terminology by making use of the thesaurus when concept descriptions are installed in the PRESS and indexed for future retrieval; these static links are made when the questionnaire is processed and entered into the database through the indexing process. This accommodates the questionnaire filler who uses non-preferred terminology in describing a software concept; the questionnaire's original content is left unaltered, but the preferred terminology of the domain is used to index the software concept. Thus a reuser may retrieve the concept knowing either the preferred or the non-preferred terminology. In a mature domain, there is little variation in terminology; but in less mature domains, it is important to accommodate such variation.

A more detailed description of the process of indexing questionnaires developed by the project and its implementation can be found elsewhere [109, 108, 6, 8]. The indexing process is semi-automated; a human index is still considered essential.

Dynamic links are made during a search; the PRESS makes active use of the thesaurus through an extension to CCL described above: the thesaurus operator. The thesaurus operator enables search terms to be augmented by additional terms based on standard thesaurus relations, such as *broader terms*, *narrower terms*, etc. This extension allowed experimentation with a form of automated search where augmentation is a function of success or failure in searching. For example, if too many concepts are retrieved using a particular term, then the PRESS could automatically "narrow" the search term, or if too few concepts are retrieved, the term could be automatically "broadened".

The Role of Facets Supporting Domain Views

Within a particular domain, the terminology may be quite diverse; and here more is at work than simply variation in terms. The diversity may reflect several viewpoints within the domain; three that were identified in the domain of Steel Production are the data processing view, the organisational view, and the application view. This resulted in three sets of terms reflecting these viewpoints. Within a particular viewpoint, it may be helpful to divide the terms used further into sets of terms describing facets of the viewpoint. The nature of these viewpoints and associated facets of terms will depend on the domain. In Figure A.3, the application viewpoint and its facets found as a result of the project studies in the domain of Steel Production are illustrated. In the Steel Production domain, the application view has the following facets: processes, end-products (materials), and equipment.

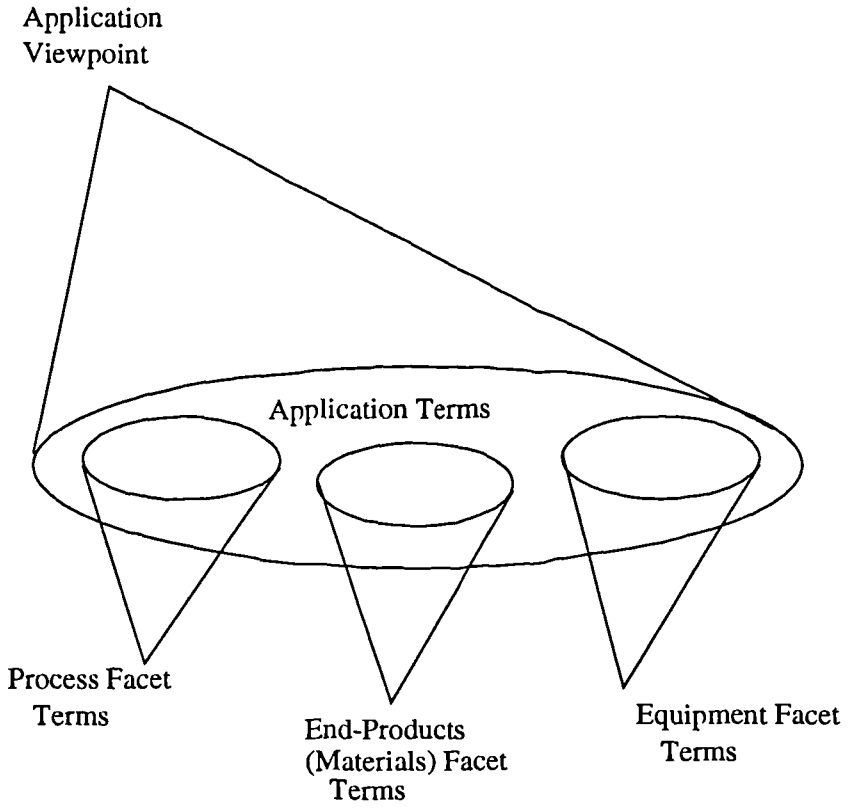


Figure A.3: Faceted Classification

Software concepts described from the application viewpoint will use terms from these facets. If every term in the thesaurus has an associated view and facet, then for any chosen view, all its terms can be systematically displayed grouped together

by facet. Using a given order of facets, say processes, end-product, equipment, all possible combinations of terms under these facets can be listed. For any given software concept, there will be a combination of terms best reflecting its description. Thus, a systematic display of the domain can be made using the relations amongst its terms and associated software concepts. This procedure is described below in greater detail.

Systematic Classification of Concepts Based on a Facetted Thesaurus

The domain of interest is considered from various viewpoints. In the case of Steel Production, three viewpoints were identified:

- Application
- Operational
- Data Processing.

Terms used within each viewpoint were grouped under facets. In the this case, the facets of the Application viewpoint are as follows: Processes, End-Products (Materials), and Equipment. In addition, these facets must be given a citation order, for example, Process followed by End-Product followed by Equipment. In this case, the citation order is the order of listing in the PRESS thesaurus.

For example, where the thesaurus contains the following entries:

```
Processes
   Forming
   Cutting
   Coating
   ...
End-Products (Materials)
   Semi-finished products
```

```
   Finished products
   ...
Equipment
   Hot rolling mills
   Cold rolling mills
   Pickling lines
   ...
```

by systematically listing every combination of these terms in citation order, the following classification headings can be obtained:

```
Forming/Semi-finished products/Hot rolling mills
Forming/Semi-finished products/Cold rolling mills
Forming/Semi-finished products/Pickling lines
Cutting/Semi-finished products/Hot rolling mills
Cutting/Semi-finished products/Cold rolling mills
Cutting/Semi-finished products/Pickling lines
Coating/Semi-finished products/Hot rolling mills
Coating/Semi-finished products/Cold rolling mills
Coating/Semi-finished products/Pickling lines
Forming/Finished products/Hot rolling mills
Forming/Finished products/Cold rolling mills
Forming/Finished products/Pickling lines
Cutting/Finished products/Hot rolling mills
Cutting/Finished products/Cold rolling mills
Cutting/Finished products/Pickling lines
Coating/Finished products/Hot rolling mills
Coating/Finished products/Cold rolling mills
Coating/Finished products/Pickling lines
```

Now we consider a particular software concept and its associated indexing terms; where the terms used to index a concept match against a particular combination of

terms under a set of facet terms, we slot in the concept name. For example, *Cold rolling* is a narrower term for *Forming*, *Cold strip* is a narrower term for *Finished products*, so the concept *Control system for cold strip rolling line* would be slotted in under the heading:

`Forming/Finished products/Cold rolling mills.`

If each concept in turn is categorised in this way a systematic display of the possible categories with actual concept occurrences is obtained; in effect, this gives a catalogue of concepts.

The process described enables a static display to be made reflecting the state of the concept base and categories based on the facets and terms recorded in the thesaurus at a particular time.

A.2.3 Practitioner METAMODEL of Reuse

The need to formulate a full metamodel for the reuse of software concepts arose out of work on a simple model of the reuse process. Work on the metamodel sought to model the common elements found in various existing approaches to reuse through the development of a higher level generalised model of reuse to be supported by the PRESS. This metamodel guided the project's work and was a principal source of reference in the development of the prototype PRESS [73]. The metamodel was defined using a conventional broad spectrum analysis and design technique, SSADM. The context of the PRESS consists of three distinct groups:

- operational organisations - sources of old software and users of software.
- sales and marketing departments/development departments - producers of requirements of new software, and
- domain experts - sources of expertise in application domains.

Although these groups may coincide in a particular organisation using the the PRESS, recognising the different contributions and requirements of each group has been one of the most important outcomes of this work. Figure A.4 reproduced from Hall's paper is the Level 0 Data Flow Diagram of the metamodel giving the top-level view. The additional levels of the metamodel give details the roles of each group in populating the PRESS with concepts, building the library at the heart of the PRESS, and ultimately using the PRESS in earnest to construct new software.

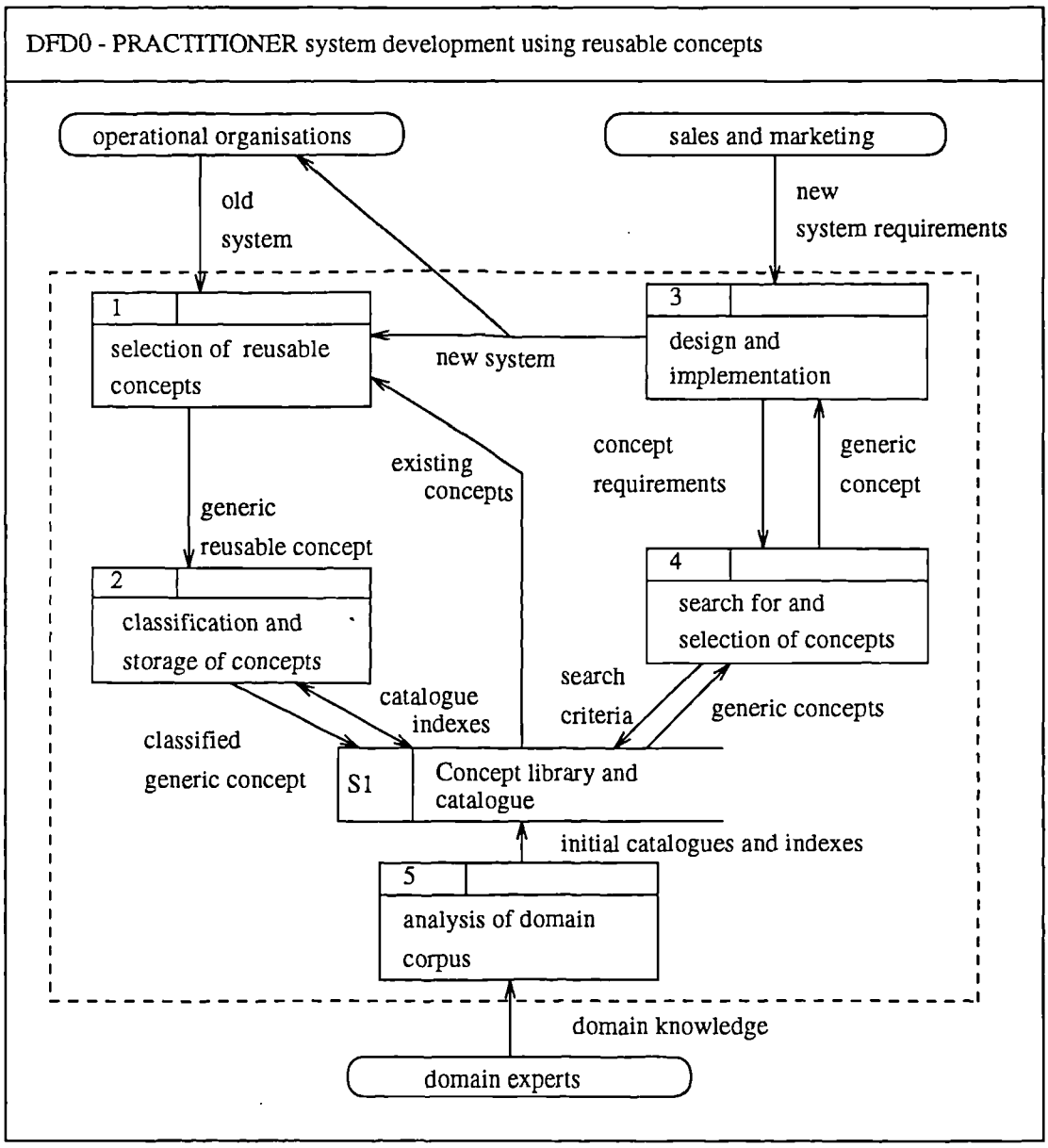


Figure A.4: Data Flow Diagram of the Practitioner Metamodel

A Reuse Support System is a variety of an information system; the technology required to support reuse is already available. The most interesting problems that the project encountered were as a result of attempts to gain a better understanding of software concept formation, the software construction process in general, and the evolution of software concepts as they are reused. In addressing these problems, the original metamodel was refined and the project formulated methods for design-for-reuse and design-with-reuse [34]. These are discussed in the following sections.

A.3 Design-for-Reuse Methods

In populating the PRESS, the project drew on the established methods of domain analysis [103, 122, 126, 129].

In the Practitioner project, particular emphasis was placed on investigating how language is used in a particular domain. This work was based on the construction of a thesaurus to record the domain terminology and studies concerning how best to use domain terminology in the indexing of software concepts to support their retrieval. The project's particular interest was the use of terms in describing software concepts in the domain, and to this end, the techniques of text analysis were applied to software application documentation ranging from requirements statements, specifications, designs to source code as well as analysing domain literature. This not only supported the work on thesaurus construction, but also it was found that such analysis provides a helpful preliminary overview of material prior to a more in depth analysis of the existing domain software.

In the construction of the thesaurus and the domain modelling and subsequent filling in of questionnaires, a large number of documents may be analysed. As many of these documents are in machine readable form, a file indexing and retrieval system to assist with the analysis process has been developed as part of the PRESS to assist in the preliminary analysis of material. This system allows sets of files to be indexed using their constituent terms or restricted subsets of these, or independently

obtained term lists such as the contents of an existing thesaurus. In the latter case, this gives a rough guide as to whether or not the document employs terminology common to the domain. This may be a useful step before using this document as the basis for filling in a questionnaire. Using the file indexing and retrieval system, sets of documents with various indices may be investigated. The user may either select sets of indexing terms with the system determining which files contain any of these terms with the possibility of viewing any selected term in context, or select a subset of files and the system will determine what indexing terms these files have in common. This system is useful in itself in organising large sets of files systematically on the basis of their contents. Here of course the organisation of files is merely a step in their more detailed analysis.

The analysis of software uses the same source of material as text analysis augmented by expertise in software engineering to gain an understanding of the domain's software concepts. Here an intermediate step corresponding to thesaurus construction, but less well understood is the modelling of the domain. The main tasks involved in domain modelling are as follows:

- study existing systems and identify concepts, first study standard design decompositions, if any;
- identify concepts that embody the structuring principles, i.e. the common levels of abstraction used, e.g. ISO 7 Layer Model for Open Systems Interconnection, Purdue Laboratory for Applied Industrial Control (PLAIC) 4 Level Model for hierarchical control systems in steel mills, etc;
- identify principal component concepts within design structures.

Output from domain analysis may take the form of modelling typical domain applications, developing a design framework for relating typical designs, or if possible developing generic architectures for well understood application families in the domain. These outputs are used to guide the domain analyst working with domain experts and software engineers in the systematic description of the domain's software concepts.

A data flow diagram of the domain analysis process used to populate the PRESS can be found in Figure A.5. Following an analysis of the domain, two results are available for populating the PRESS - a thesaurus relating domain terminology and a collection of related domain concepts described in the form of sets of questionnaires. The relation between these two results is that the terms in the thesaurus derive from the same domain language as that used in describing the domain's software concepts. As new software concept descriptions are added, the thesaurus is employed in their indexing to uplift any use of non-preferred vocabulary. At the same time, as new software descriptions are obtained, their text is analysed for the possibility of determining new candidate terms for entry into the thesaurus. So the development of the thesaurus is related symbiotically to that of the software concept database.

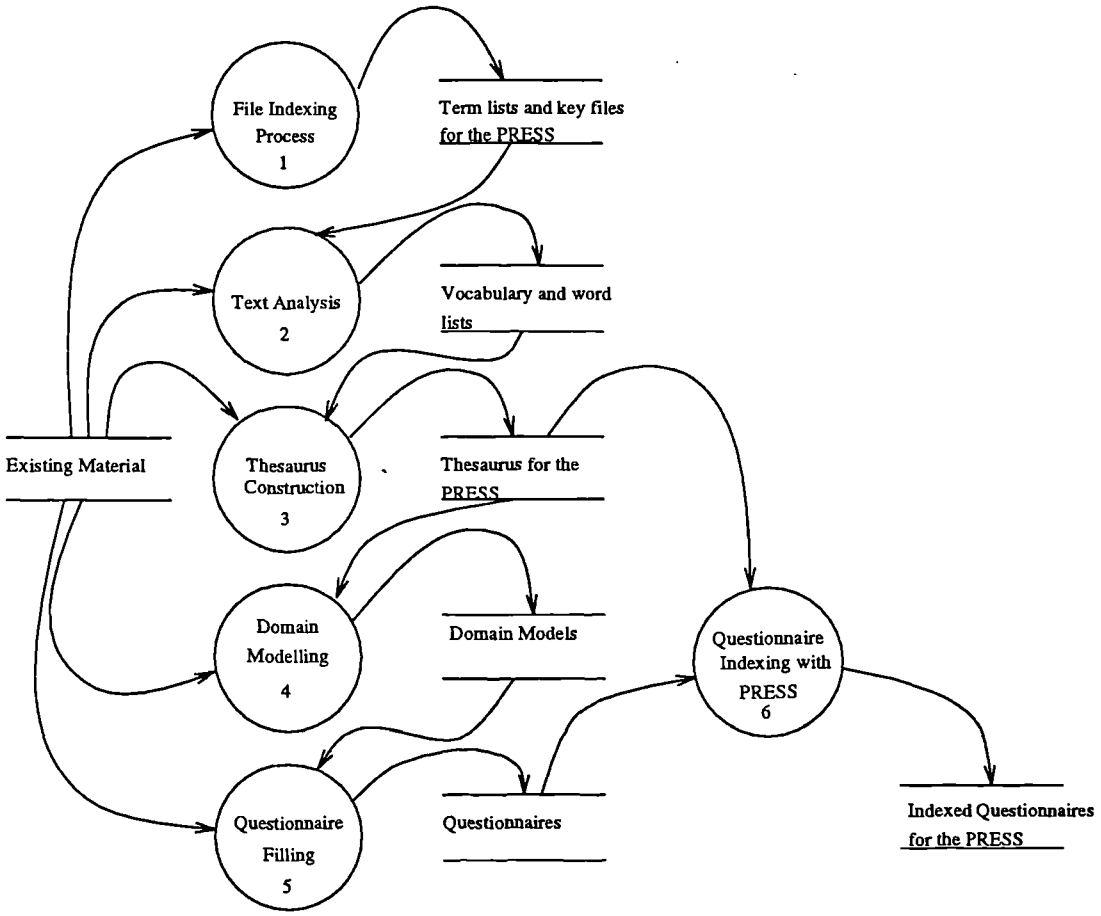


Figure A.5: Data Flow Diagram of the Domain Analysis Process Used to Populate the PRESS

A.4 Design-with-Reuse Methods

The individual designer using the PRESS whatever specific design approach is employed is faced with three primary tasks, summarised as cognition, conceptualisation and construction. The cognition task involves understanding and clarification of the requirements. Where these take the form of a text, the designer may draw on the deconstructionalist approach to logical understanding of a text (described in [105], breaking the text down into its constituent terms and mapping these onto known terms in the PRESS thesaurus. It has been suggested that the development of a problem taxonomy (or taxonomies), i.e. a rudimentary data dictionary, is a useful exercise for the designer at this stage, and here the PRESS, if populated with suitable terminology and concepts, is a valuable aid. In the PRESS, the terminology recorded in the thesaurus plays an important role in supporting the designer during the initial task of conceptualisation in searching for relevant solution concepts to satisfy the requirements. The terms held in the thesaurus are used in indexing the concept descriptions, i.e. questionnaires; and the designer is able to explicitly make use of thesaurus relations in searching the questionnaire database for design concepts. For example, the designer may elect to extend the search for concepts by including "broader terms" of a given search term.

Having retrieved a set of potentially relevant design concepts using selected terms from the requirements statement and related terms from the thesaurus, the designer will need guidance on how to make use of the retrieved material to compose a new design to meet the given requirements. Three main steps are appropriate to assist the designer; they are as follows:

1. identify the key concepts in context by viewing their relations to other domain concepts held in the PRESS,

2. determine the design structure either through identification of a relevant architectural concept or by identifying appropriate layers based on requirements,

3. identify the relevant component concepts that satisfy the functional and non-

functional requirements.

Clustering the requirements using Alexander's method may be helpful during the second step [9]. More generally, the guidelines of Zimmerman are relevant in order to determine the appropriate levels of conceptual abstraction to be employed in the design [176].

The degree of reuse possible in the context of the PRESS will of course depend on the success with which the designer is able to make the above identifications within the confines of a particular population of the PRESS. Here it cannot be denied that creativity is needed to make such identifications, but through the provision of the PRESS, the designer is given assistance with the important task of conceptualisation.

Through the informal semantics of the interfaces given in the concept descriptions, the designer is able to carry out informal interface checking at the conceptual stage. The PRESS also is able to give the designer some assistance with the task of construction although this is limited to the optional links to more detailed design and implementation descriptions that may be found in a particular questionnaire. Finally, once the new design has been composed, the PRESS can be used to check whether or not its overall usage of terminology is consistent. This step has the direct benefit of improving the expression of the design and the added benefit of being the first stage of analysing the new design description for incorporation into the PRESS for reuse in future. These steps are summarised in Figure A.6 - Data Flow Diagram of Design Process with the PRESS.

Figure A.7 - PRESS Users - which provided the *leitmotiv*, i.e. controlling theme, of the project review in May 1991 brings together the PRESS with its intended users.

A.4.1 The Role of Design Frameworks

The approach of using design frameworks was proposed based on initial experience of analysing software concepts in the Steel Production and on insights gained from

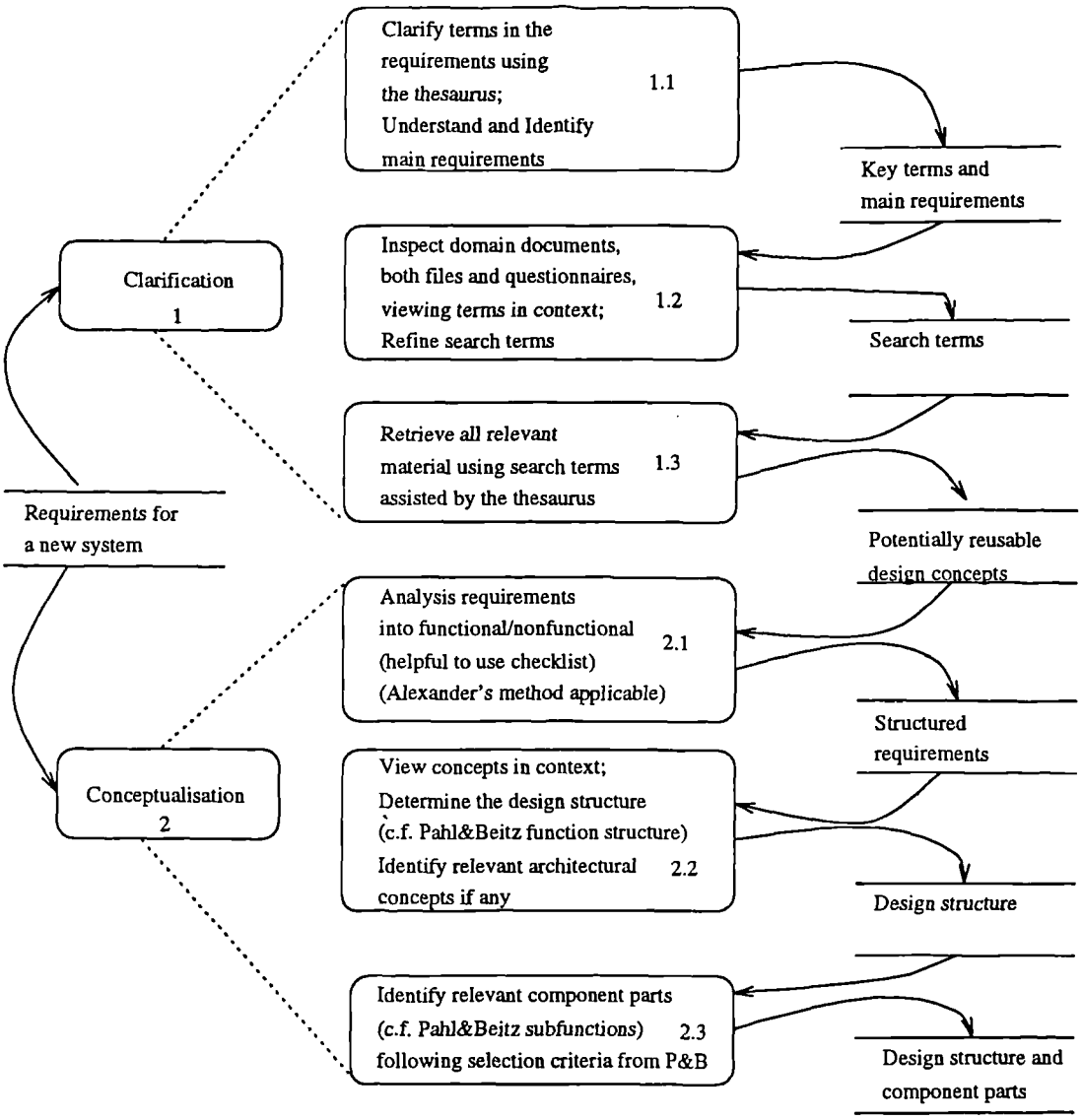


Figure A.6: Data Flow Diagram of Design Process with the PRESS

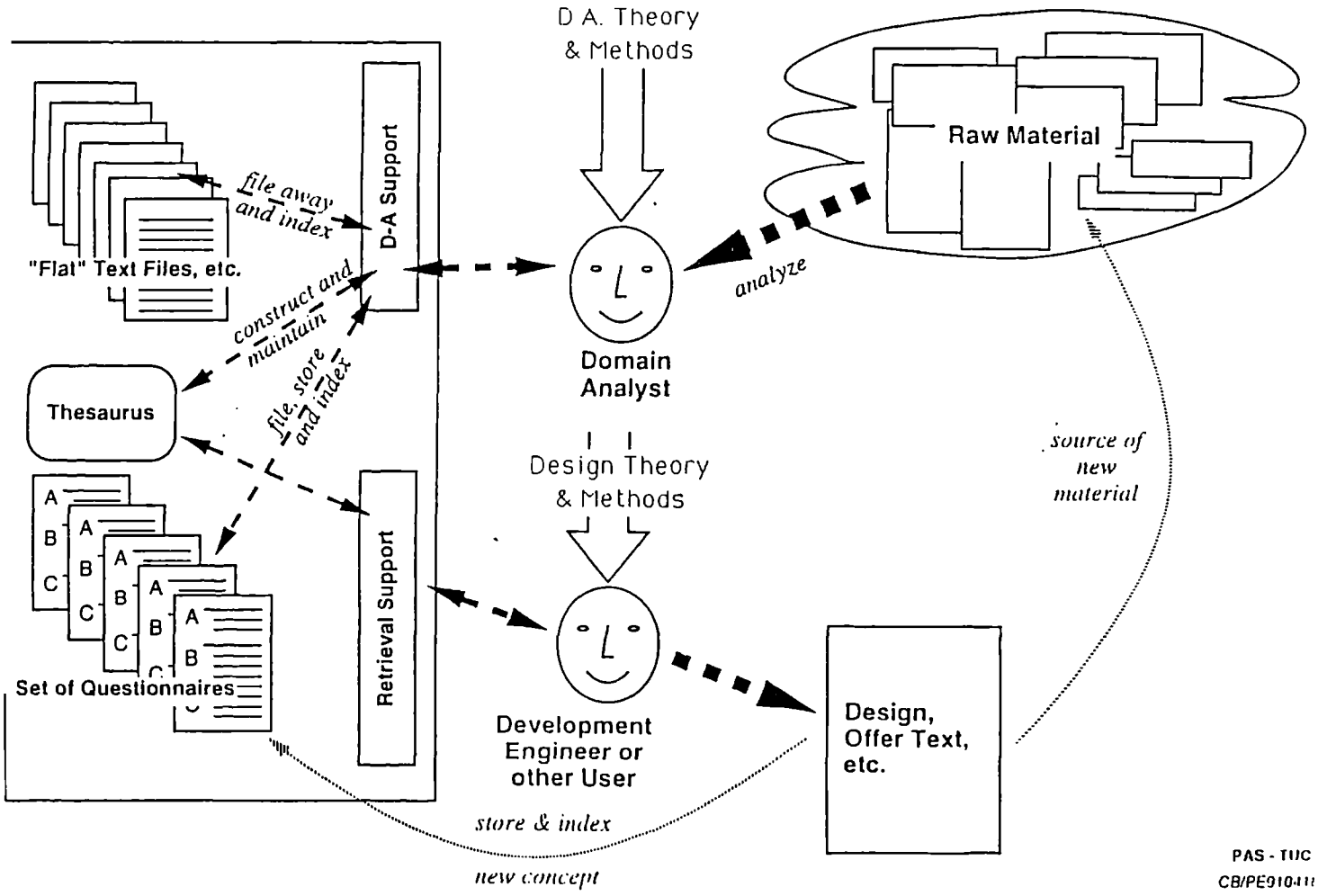


Figure A.7: PRESS Users

considering how the concepts identified could be related to best support the design-with-reuse process [30, 29]. The opportunity to refine it through application in the course of the project was provided by further studies described in [32]. This research benefited from applying the methods and principles of *Konstruktionslehre*, i.e. Construction Theory, as developed in [110]. Through their application, it was possible to develop a more systematic approach to domain analysis of existing software designs; the approach of design frameworks supports *systematic design-for-reuse in the large*. This will be demonstrated here in Chapter 5. The development and population of a design framework provides a mechanism for structuring the work of the domain analyst, within which more detailed analysis of software concepts can take place. The design understanding recorded in the framework is itself an important reusable result. Design frameworks provide a conceptual frame within which the reusable design concepts of a domain can be related. In attacking a design problem, the development engineer can benefit directly by being able to reuse the design understanding and experience recorded in the framework and the concepts it relates. These remarks anticipate the conclusions found in Chapter 6 of this thesis.

A.5 Author's Contribution to the Practitioner Project

This thesis draws on the author's contribution to work within the Practitioner project. In particular, it is based on the research carried out by the author in the following areas:

- refinement of the questionnaire to describe software concepts,
- development of methods to support design-with-reuse and design-for-reuse,
- studies of designs in the domain of Steel Production and
- development of PRESS demonstrations.

The following sections gives details of the author's contributions to the project in terms of work package deliverables, working papers and associated external publications.

A.6 Listings of Author's Contributions to the Practitioner Project

A.6.1 Internal Publications

The author's main contributions to the project can be found in the following work package deliverables:

A1.1 Literature survey on descriptive methods

A1.2 Recommendations on the use of descriptive methods

B1.1 Rules and guidelines for the application of the descriptive method

C1.1 Specification of support system prototype

D2.1 Interim project report and evaluation

E2.4 Demonstration of the PRESS

E4.1 Report on 'concept and module surfaces'

E4.2 Handbook for design of 'concept and module surfaces'

F4.1 Research report on concept isolation

F4.2 Handbook on application of principles

and in the following working papers written by the author:

BrU-0001 C4: Program Linguistics

BrU-0009 Descriptive Methods Survey and Recommendations

BrU-0011 Proposal for Investigations Concerning LIL

BrU-0012 C1.1 Section on User Interfaces

BrU-0022 Investigations Concerning Concept Representation

BrU-0033 CAMP Project Summary

BrU-0036 Reuse, Software Concepts, Descriptive Methods and the Practitioner Project

BrU-0050 The Role of Text Analysis in Software Reuse

BrU-0056 The Questionnaire: A generic form for the description of software concepts

BrU-0062 Revision of the Questionnaire: an Update on Progress

BrU-0071 Software Classification

BrU-0084 Relationship Between Questionnaire and Thesaurus

BrU-0102 Detailed Plan for the Demonstration of the PRESS

BrU-0112 Proposal For a Strategy To Support Questionnaire Browsing

PAS(BrU)-1 Proposal for Further Work on Design-with-Reuse

PAS(BrU)-2 The PRESS A Consideration From Tool Interconnection

PAS(BrU)-3 Outline of Pahl and Beitz's Konstruktionlehre

PAS(ULi)-7 Notes on Background Work on Design-with-Reuse

PAS(ULi)-9 Report Following First Visit to Salzgitter Steel Mill

PAS(ULi)-11 A review of Design Problem Solving

PAS(ULi)-12 Displaying Concepts by Unfolding

PAS(ULi)-13 A Framework for Software Concepts in Steel Production.

A.6.2 External publications

In addition, there are a number of external publications which have resulted from the author's work on the project; a list of these is given below:

1. C Boldyreff, *Reuse, Software Concepts, Descriptive Methods and the Practitioner Project*, pp 25-31, ACM SIGSOFT Software Engineering Notes, Vol 14, No 2, April 1989.

2. C Boldyreff and P Hall, *Reusablity: Evaluation of Practitioner Project Experience and Future Directions*, CASE 89 Advance Working Papers, pp 470-471, Imperial College, London, July 17-21, 1989.

3. C. Boldyreff, P. Elzer, P. Hall, U. Kaaber, J. Keilmann and J. Witt, *PRACTITIONER: Pragmatic Support for the Reuse of Concepts in Existing Software*, Proceedings of Software Engineering 1990, Brighton, UK, Cambridge University Press, 1990.

4. C Boldyreff, P Hall and J Zhang, *Reusability: The Practitioner Approach*, Position paper printed in: Workshop "Reuse" RESEARCH IN PROGRESS, Delft University of Technology, pp 1-9, November 1989.

5. P Hall and C Boldyreff, *Software Reuse Overview*, Technical Briefing, Paper printed in the Proceedings of REUSE, MAINTENANCE AND REVERSE ENGINEERING OF SOFTWARE: Current Practice and New Directions, UNICOM Seminars Limited, London, 29 November-1 December 1989.

6. C Boldyreff and J Zhang, *FROM RECURSION EXTRACTION TO AUTOMATED COMMENTING - A Transformational Approach towards Reverse Engineering of Software to Support Reusability*, Paper printed in the Proceedings of REUSE, MAINTENANCE AND REVERSE ENGINEERING OF SOFTWARE: Current Practice and New Directions, UNICOM Seminars Limited, London, 29 November-1 December 1989.
(Reprinted as: C. Boldyreff and J. Zhang, *From recursion extraction to automated commenting*, in **Software Reuse and Reverse Engineering in Practice**, Edited by P. A. V. Hall, pp 253-270, Chapman & Hall, 1992.)

7. Patrick Hall and Cornelia Boldyreff, *Software reuse*, in **Software Engineering Reference Book**, John McDermid, Editor, Butterworths, June 1990.

8. Kim Bisgaard, Cornelia Boldyreff, Peter Elzer, Pat Hall, Johannes Keilmann, Horst Kern, Lene Olsen, Jan Witt and Jian Zhang, *The Practitioner REuse Support System (PRESS): A Tool Supporting Software Reuse*, Proceedings of the Third Annual Workshop: Methods and Tools for Reuse, CASE Center, Syracuse University, 13-15 June 1990.

9. Jian Zhang and Cornelia Boldyreff, *Towards Knowledge-Based Reverse Engineering*, Proceedings of the Fifth Annual Knowledge-Based Software Assistant Conference, Syracuse, NY, 24-28 September 1990.

10. Cornelia Boldyreff, *Supporting System Design From Reusable Design Frameworks*, Proceedings of the Second International Conference on INFORMATION SYSTEM DEVELOPERS WORKBENCH Methodologies, Techniques, Tools and Procedures, Gdansk, 25-28 September 1990.
(Reprinted in an up-dated version as Cornelia Boldyreff, *Design methods for integrating system components*, in **Software Reuse and Reverse Engineering in Practice**, Edited by P. A. V. Hall, pp 81-97, Chapman & Hall, 1992.)

11. Cornelia Boldyreff, *A CASE Tool Supporting Reuse: the PRESS*, CASE '90 Fourth International Workshop on CASE, Irvine, CA, USA, 5-8 December 1990.

12. Pat Hall, Cornelia Boldyreff and Jian Zhang, *PRACTITIONER: Pragmatic Support for the Re-use of Concepts in Existing Software*, in **Software REuse, Utrecht 1989**, Liesbeth Dusink and Patrick Hall (Eds.), Workshops in Computing Series, Springer-Verlag, 1991.

13. Cornelia Boldyreff, *What can Software Engineering learn from Design Theory?*, Position paper produced for discussion panel on Design Activity and Design Issues at the CSCW-SIG Workshop on Design Issues in CSCW, DTI, London, 17 March 1992.

14. Cornelia Boldyreff and Uwe Krohn, *The Practitioner Reuse Support System (PRESS): A Consideration from the Standpoint of Tool Interconnection*, Proceedings of the Fourth IFAC/IFIP Workshop on Experience with the Management of Software Projects, Austria, May 18-19, 1992. The proceedings have been published as a book by Pergamon Press.

15. Cornelia Boldyreff, *A Design Framework for Software Concepts in the Domain of Steel Production*, Proceedings of the Third International Conference on Information System Developers Workbench, Gdansk, 22-24 September 1992.

16. Cornelia Boldyreff, *Design Frameworks: A Basis for Recording Program Understanding*, Position paper for IEEE Workshop on Program Comprehension. Orlando, Florida, 9th November 1992.

Appendix B

Listings of Main Headings of Practitioner Project Questionnaire

```
Name of Concept:
Name of File:
Reported by:
Institution:
Date:
Classification:
Keywords:
A. Application Oriented Description
A.1 Textual Description
A.2 Graphical Description
B. Implementation Oriented Description
B.1 Requirements Level
B.1.1 Functional Description
B.1.2 Reactions to Exceptions
      Exception Condition                Expected Reaction
B.1.3 Data Input (Number, Type, Frequency, Range, etc.)
B.1.4 Data Output (Number, Type, Frequency, Range, etc.)
B.1.5 Control Input (e.g. User commands, signals etc.)
B.1.5 Control Output
B.2 Design Level
B.2.1 Used Functions (e.g. software functions necessary)
B.2.2 Immediate Parts (IPs)
B.2.2.1 Active Immediate Parts
        Name (Atomic (Y/N))   Short Characteristics
        References to Descriptions of Non-atomic Parts
B.2.2.2 Passive Immediate Parts
        Name (Atomic (Y/N))   Short Characteristics
        References to Descriptions of Non-atomic Parts
B.2.3 Relations between Immediate Parts
```

B.2.3.1 Static Relations
B.2.3.2 Dynamic Relations (e.g. calling sequence of IPs or parallelism)
B.3 Implementation Level
B.3.1 Immediate Parts
B.3.1.1 Active Immediate Parts
Name Implemented by
B.3.1.1 Passive Immediate Parts
Name Implemented by
B.3.2 Relations between Immediate Parts
B.3.2.1 Data flow (e.g. global section, file, database, mailbox)
From To Mechanism Data flow (name)
B.3.2.2 Control flow (e.g. event, semaphore, mailbox)
From To Mechanism Control flow (name)
B.4 Source Code Level
B.4.1 Programming Language
B.4.2 Size of source program
B.4.3 Operating System
B.4.4 Type of computer(s)
B.4.5 Technical requirements of computer if available
(e.g. memory size, background storage, etc.)
C. Historical Development
C.1 From which project
C.2 When developed
C.3 Developer
C.4 Version Number
C.5 Origin of relevant concepts
C.6 Respective documentation (type and location)
C.7 Further documentation at same level (type and location)
C.8 Further documentation on related functions (type and location)

Appendix C

Abstract Syntax of the CDF and Informal Semantics of CDF Entries

C.1 Abstract Syntax of the CDF

In BNF, the CDF has the following abstract syntax:

```
CDF ::= Concept_Version_and_Derivation
                  Concept_Specification
                  Concept_Decomposition_OPTION
                  Concept_Links_OPTION

Concept_Version_and_Derivation ::=
                  Concept_Name
                  Version_Number
                  Derivation_OPTION

Concept_Specification ::=
                  Definition
                  Interfaces_Provided
```

```
                Interfaces_Required

Concept_Decomposition ::=
                Concept_Component_LIST

Concept_Links ::=
                Code_Module_OPTION
                Data_Definition_OPTION
                Documentation_OPTION
                Test_Package_OPTION

Concept_Name ::= <Unique Name>

Version_Number ::= <Unique Number>

Derivation ::=  Description_of_Purpose
                Authorising_Person
                Created_by_Person
                Date_of_Creation_or_Entry
                Source_Concept_Versions
                Creation_Processes_Used

Description_of_Purpose ::=
                <Text of Concept Version Requirements Statement>

Authorising_Person ::= <Name>

Created_by_Person ::= <Name>

Date_of_Creation ::= <Date>

Source_Concept_Versions ::= Source_Concept_Version_LIST

Source_Concept_Version ::=
                Concept_Name
                Version_Number

Creation_Processes_Used ::=
                <Text describing creation processes>

Definition ::=  Function_OPTION
                Formalism_OPTION
                Generic_Parameters_OPTION
                Description
```

```
Interfaces_Provided ::=
                Interface_LIST

Interfaces_Required ::=
                Interface_LIST

Interface ::= Concept_Name
                 Version_Number

Concept_Component ::=
                Concept_Being_Instantiated
                Instantiation_Parameters_or_Specialisations_OPTION
                Purpose_Served_or_Reason_for_Incorporation_OPTION
                Interface_Bindings

Function ::= <Text of Concept Functional Specification>

Formalism ::= <Name>

Generic_Parameters ::=
                <Text describing generic parameters>

Description ::= <Text of Informal Concept Specification>

Concept_Being_Instantiated ::=
                 Concept_Name
                 Version_Number

Instantiation_Parameters_or_Specialisations ::=
                 <Text describing parameters or specialisations>

Purpose_Served_or_Reason_for_Incorporation ::=
                 <Text de3scribing purpose or reason>

Interface_Bindings ::=
                External_Concept_Interface_Binding_LIST
                Internal_Concept_Interface_Binding_LIST

External_Concept_Interface_Binding ::=
                 Interface
                 ->
                 Interface

Internal_Concept_Interface_Binding ::=
                 Interface
```

```
                    ->
                    Interface

Code_Module ::= <document reference or filename>

Data_Definition ::= <document reference or filename>

Documentation ::= Document_LIST

Document ::= <document reference or filename>

Test_Package ::= <document reference or filename>
```

Note that all productions ending with LIST and SEQUENCE are simply macro expanded using the hyper-rules:

```
Notion_LIST ::= Notion | Notion_LIST Notion

Notion_SEQUENCE ::= EMPTY | Notion | Notion_SEQUENCE Notion
```

Rather than using the convention of indicating optional notions using square brackets, the hyper-rule given below has been employed:

```
Notion_OPTION ::= Notion | EMPTY
```

The use of hyper-rules here follows that given in the Revised Report on Algol 68 [161].

C.2 Informal Semantics of CDF Entries

In what follows, one concrete form of the CDF is listed below with annotations which informally specify what it expected under each heading. This was prepared for the benefit of engineers concerned with filling in CDFs during the Practitioner project. Some details about how the information about a concept is held in the PRESS concept database are also included where these are considered relevant.

```
Concept Descriptors
<indexing terms for this CDF>
Date of Indexing
<date in form dd-mmm-yy, e.g. 20-SEP-90>
```

Note: This is information held regarding a particular CDF and is not considered as part of the contents of a CDF. In any case, the indexing of the CDF is unlikely to be done by the filler-in of its contents. In the PRESS, this process is automated although allowance has been made for an expert to override this.

```
Language Code
<code>
Keywords
<list of keywords>
```

Note: As with the above information, this is not part of the CDF contents; it must be supplied by the CDF installer prior to parsing before a particular CDF can be entered into the PRESS database.

```
1. Concept Version and Derivation
1.1 Concept Name
<name of concept>
```

Explanation: This is the unique name of the concept. For example, it could be **Hot Mill Unit Control and Supervision.**

```
1.2 Concept Version
<version number: integer>
```

Explanation: This is number allows more than one version of a concept to be described and held in the PRESS.

Note: Both the name of the concept and its version number are used to identify a particular CDF within the PRESS. Whilst these may be given by fillers-in of the CDF, the PRESS administrator needs to exercise control of this information to ensure the integrity of the PRESS name space. The concept name and version number are not strictly part of the CDF

contents employing the distinction mentioned above between information "regarding the CDF" and information constituting "the contents of the CDF". This distinction was proposed by Kim Bisgaard of CRI, one of the principal developers of the PRESS.

1.3 Derivation
1.3.1 Description of Purpose
<requirements giving rise to this concept>

Explanation: This section records the requirements that gave rise to the concepts' development.

1.3.2 Authorising Person
<name of person>

Explanation: If appropriate, for all new concepts and versions of existing concepts that are installed in the PRESS, an authorising person is identified.

1.3.3 Created by Person
<name of person>

Explanation: Although of historical interest, if known, the creator of the concept description should be recorded.

1.3.4 Date of Creation or Entry
<date in form dd-mmm-yy, e.g. 20-SEP-90>

Explanation: This date gives guidance on the currency of concept descriptions and allows development of the PRESS over time to be monitored.

1.3.5 Source Concept Versions
<references to concepts from which this concept is derived>

Explanation: This entry allows the *derived from* relation to be established between an new concept and existing concepts. This relation is important for understanding the intellectual development of a concept. Typically these will be references to other concept descriptions.

1.3.6 Creation Processes Used
<references to processes used in the creation of this concept>

Exaplanation: The intellectual processes leading to the development of a new concept such as generalisation or specialisation should be noted here. Tools used such a program generator or 4GL should be mentioned. These may well be references to other concept descriptions.

2. Concept Specification
2.1 Definition
2.1.1 Function
<functional specification>

Explanation: This is the specification in the traditional Software Life Cycle sense. It will usually be given using a particular specification method and associated formalism.

2.1.2 Formalism
<formalism used in above functional specification>
2.1.3 Generic Parameters
<description of any parameterised aspects of the concept>

Explanation: The parameters and other variables that need to be bound when a generic concept is instantiated for a particular usage.

2.1.4 Description
<informal application oriented description of concept>

Explanation: This is the less formal description of the concept. It was previously known as the Application Oriented Description. Typically it will use terminology specific to the application domain. It gives an account of what role of the software concept plays in the context of a particular software application.

2.2 Interfaces Provided
2.2.1 Interface Name
<interface name>
2.2.2 Interface Version
<version number>

Explanation: Here the concept's points of connection or interaction that it provides as services for other concepts are listed by name and version number pairs. These interfaces may be functions, shared data, exceptions, etc; and may of course also have their own associated CDF.

Note: This whole set of headings is repeated for each interface provided.

```
2.3 Interfaces Required
2.3.1 Interface Name
<interface name>
2.3.2 Interface Version
<version number>
```

Explanation: A concept may require the services of other concepts. Here such interfaces required by the concept are listed. As with interfaces provided, the entries here may be references to other CDFs. Where it is unclear whether an interface is provided or required, it should be recorded as required.

Note: This whole set of headings is repeated for each interface required.

```
3. Concept Decomposition
3.1 Component Concept
3.1.1 Concept Being Instantiated
3.1.1.1 Concept Name
<name of concept>
3.1.1.2 Concept Version
<version number>
```

Explanation: The concept name and concept version number of each component concept together can be used to identify the concept being instantiated as one that may already be described by an existing CDF.

```
3.1.1.3 Instantiation Parameters or Specialisation
<parameters/specialisation values used in this instance>
```

Explanation: If the particular subconcept being instantiated, already exists in the concept database in a generic form, then the values used in the act of instantiation need to be recorded here.

3.1.1.4 Purpose Served or Reason for Incorporation
<brief description above>

Explanation: A particular concept may have been selected to serve some particular purpose or for some explicit reason within the decomposition. This should be recorded here. This may be used to give an explanation for why a particular concept decomposition has been made.

3.2 Interface Bindings
3.2.1 External Concept Interface Bindings
<list of bindings>

Explanation: These interface bindings link actual subconcept interfaces with interfaces provided or required.

3.2.2 Internal Concept Interface Bindings
<list of bindings>

Explanation: These interface bindings establish internal links between the subconcept interfaces.

> Note: This whole set of headings is repeated for each immediate part of the concept.

4. Links
4.1 Code Module
<document reference or filename>

Explanation: This is a reference to the source code that implements a concept, typically a document reference or filename.

4.2 Data Definition
<document reference or filename>

Explanation: This is a reference to any relevant data definition document, e.g. data dictionary.

4.3 Documentation
<document reference or filename>

Note: This heading is repeated for each reference. These could be document references or names of files containing documentation.

4.4 Test Package
<document reference or filename>

Explanation: This is a reference to any test package to be used with the concept.

Appendix D

An Example CDF

Note that where a Concept Name and Version Number are given, this has been abbreviated in most cases to simply the proper name followed by a hash symbol and the version number. E.g.
Pickling and Cold Mill Control and Supervision#1
instead of
Concept Name:Pickling and Cold Mill Control and Supervision
Version Number: 1

D.1 CDF describing the software concept of a Tandem Mill Automation Scheme

1. Concept Version and Derivation
1.1 Concept Name: TAMS Tandem Mill Automation Scheme
1.2 Version Number: 1
1.3 Derivation
1.3.1 Description of Purpose:

This tandem mill automation scheme attacks the problem of determining the correct mill set-up for each new coil, so as to obtain smooth operation, high product quality, and to minimize the need for operator intervention.

1.3.2 Authorising Person:

1.3.3 Created by Person:

Cornelia Boldyreff

1.3.4 Date of Creation or Entry:

30-June-91

1.3.5 Source Concept Versions:

Pickling and Cold Mill Control and Supervision#1

1.3.6 Creation Processes Used:

2. Concept Specification

2.1 Definition

2.1.3 Description:

> A block diagram of the basic functional structure and components is given in Reference 1: Bryant's Automation of Tandem Mills, Figure 1, Chapter 8, page 142.
>
> Tandem Mill Automation is concerned with determining the correct mill set-up for each coil. Mill set-up can be described in terms of three conceptually distinct phases: Nominal Schedule Calculation, Schedule Adaption and Schedule-Dependent Gain Calculation.
>
> In the first phase, a nominal schedule is calculated from the nominal characteristics of the next coil to be rolled, and in the absence of any disturbances, this schedule will be used. In practice, the mill is subject to significant disturbances, and hence the need for phase two in which on-line adaptation of the schedule takes place. The final phase is needed to calculate the actuator references to achieve the specified schedule.

2.2 Interfaces Provided

TAMS.Mill Actuator References (motor speed, screw position, jack force)#1

2.3 Interfaces Required

TAMS.Coil Characteristics (strip width, mill entry gauge, exit gauge, strip grade)#1
TAMS.Plant Measurements#1
TAMS.Maximum Mill Speed (operator input)#1
TAMS.Load Adjustment (operator input)#1
TAMS.Shape Corrections (operator input)#1
TAMS.Tension Adjustments (operator input)#1
TAMS.Thickness-gauge Fault Factors (operator input)#1
TAMS.Actuator Adjustments (operator input)#1

3. Concept Decomposition

3.1 Concept Component

3.1.1 Concept Being Instantiated:

NSC Nominal Schedule Calculation#1

3.2 Interface Bindings
3.2.1 External Concept Interface Bindings:
NSC.Coil Characteristics (input)#1 → TAMS.Coil Characteristics (strip width, mill entry gauge, exit gauge, strip grade)#1
3.2.1 Internal Concept Interface Bindings:
NSC.Nominal Schedule (output)#1 → SA.Nominal Schedule (input)#1
3.1 Concept Component
3.1.1 Concept Being Instantiated:
SA Schedule Adaption#1
3.2 Interface Bindings
3.2.1 External Concept Interface Bindings:
SA.Maximum Mill Speed (input)#1 → TAMS.Maximum Mill Speed (operator input)#1
SA.Load Adjustment (operator input)#1 → TAMS.Load Adjustment (operator input)#1
SA.Shape Corrections (operator input)#1 → TAMS.Shape Corrections (operator input)#1
SA.Tension Adjustments (operator input)#1 → TAMS.Tension Adjustments (operator input)#1
3.2.2 Internal Concept Interface Bindings:
SA.Nominal Schedule (input)#1 → NSC.Nominal Schedule (output)#1
SA.Estimates (input)#1 → PE.Estimates (yield stress, friction coefficient, thermal camber, screw datum error) (output)#1
SA.Adapted Schedules (Rolling and Threading) (output)#1 → SDGC.Adapted Schedules (input)
SA.Adapted Schedules (Rolling and Threading) (output)#1 → MSC.Adapted Schedules (input)
SA.Adapted Threading Schedule (output)#1 → FFCS.Adapted Threading Schedule (input)
3.1 Concept Component
3.1.1 Concept Being Instantiated:
SDGC Schedule-Dependent Gain Calculation#1
3.2 Interface Bindings
3.2.1 External Concept Interface Bindings:
3.2.2 Internal Concept Interface Bindings:
SDGC.Adapted Schedules (input)#1 → SA.Adapted Schedules (Rolling and Threading) (output)#1
SDGC.Controller Gains (output)#1 → FCS.Controller Gains (input)#1
SDGC.Controller Gains (output)#1 → FFCS.Controller Gains (input)#1
3.1 Concept Component
3.1.1 Concept Being Instantiated:
MSC Mill Set-up Calculation#1
3.2 Interface Bindings
3.2.1 External Concept Interface Bindings:
3.2.2 Internal Concept Interface Bindings:

MSC.Adapted Schedules (input)#1 → SA.Adapted Schedules (Rolling and Threading) (output)#1
MSC.Estimates (input)#1 → PE.Estimates (yield stress, friction coefficient, thermal camber, screw datum error) (output)#1
MSC.Controller References (mill entry gauge, exit gauge, exit tension stress, thread roll force on the strip front end) (output)#1 → FCS.Controller References (input)#1
MSC.Mill Actuator References (motor speed, screw position, jack force) (output)#1 → MARA.Actuator References (input)#1
3.1 Concept Component
3.1.1 Concept Being Instantiated
FFCS Feedforward Control System#1
3.2 Interface Bindings
3.2.1 External Concept Interface Bindings:
FFCS.Thread Gauge Deviations (input)#1 → TAMS.Plant Measurements (input)#1
3.2.2 Internal Concept Interface Bindings:
FFCS.Adapted Threading Schedule (input)#1 → SA.Adapted Threading Schedule (output)#1
FFCS.Controller Gains (input)#1 → SDGC.Controller Gains (output)#1
FFCS.Feedforward Control Data (output)#1 → FCS.Feedforward Control Data (input)#1
3.1 Concept Component
3.1.1 Concept Being Instantiated:
FCS Feedback Control System#1
3.2 Interface Bindings
3.2.1 External Concept Interface Bindings:
FCS.Thread Gauge Deviations (input)#1 → TAMS.Plant Measurements (input)#1
3.2.2 Internal Concept Interface Bindings:
FCS.Controller Gains (input)#1 → SDGC.Controller Gains (output)#1
FCS.Controller References (input)#1 → MSC.Controller References (mill entry gauge, exit gauge, exit tension stress, thread roll force on the strip front end) (output)#1
FCS.Feedforward Control Data (input)#1 → FFCS.Feedforward Control Data (output)#1
FCS.Feedback Control Data (output)#1 → MARA.Feedback Control Data (input)#1
3.1 Concept Component
3.1.1 Concept Being Instantiated:
MARA Mill Actuator References Adjustment#1
3.2 Interface Bindings
3.2.1 External Concept Interface Bindings:
MARA.Actuator Adjustments (operator input)#1 → TAMS.Actuator Adjustments (operator input)#1
MARA.Mill Actuator References (output)#1 → TAMS.Mill Actuator References (motor speed, screw position, jack force) (output)#1
3.2.2 Internal Concept Interface Bindings:
MARA.Feedback Control Data (input)#1 → FCS.Feedback Control Data (out-

put)#1
MARA.Actuator References (input)#1 → MSC.Mill Actuator References (motor speed, screw position, jack force) (output)#1
3.1 Concept Component
3.1.1 Concept Being Instantiated:
PE Parameter Estimation#1
3.2 Interface Bindings
3.2.1 External Concept Interface Bindings:
PE.Plant Measurements#1 → TAMS.Plant Measurements#1
PE.Strip Grade (input)#1 → TAMS.Coil Characteristics (strip grade only)#1
PE.Thickness-gauge Fault Factors (operator input)#1 → TAMS.Thickness-gauge Fault Factors (operator input)#1
3.2.2 Internal Concept Interface Bindings:
PE.Estimates(yield stress, friction coefficient, thermal camber, screw datum error) (output)#1 → SA.Estimates (input)#1
PE.Estimates(yield stress, friction coefficient, thermal camber, screw datum error) (output)#1 → MSC.Estimates (input)#1
4. Links
4.1 Code Module:
4.2 Data Definition:
4.3 Documentation:
Reference 1: Automation of tandem mills, G. F. Bryant (editor), The Iron and Steel Institute, London, 1973.
4.4 Test Package:

Appendix E

Customer Requirements for Galvanizing Line Control System - DVL2

E.1 Basic Data and Requirements for DVL2

Incoming material:

- Width : 700 - 1850 mm
- Thickness : .4 - 1.6 mm
- Material : Full hard cold rolled
- Yield Strength : 650 - 850 N/mm$\hat{2}$
- Coil Weight : 10 - 48.00 kg
- Coil Diameter : 610 - 2800 mm

Outgoing material:

- Two side coated *galvanized* (CI)
- Two side coated *galvanealed* (GA)
- Differential coating 1 : 3 maximum

- Coating weight per side: 30 - 100 gr/m$\hat{2}$
  Accuracy plus or minus 3.5 g/m$\hat{2}$
- Roughness .87 - 1.87 micrometer Ra
- Mainly : automotive
  Secondary : white goods and electrical appliances

Process equipment:

- Entry/exit section with two un/recoilers
- Chemical cleaning section
- Annealing furnace
- Two zincpots (electrical heating)
- Airknife
- Galvaneal furnace
- Temper mill
- Tension leveller
- One or more chemical after treatment section(s)
- Waste water treatment plant

Line data:

- Line speed : 150 m/min
- Max. entry speed : 225 m/min
- Max. exit speed : 260 m/min
- Accel./Deceleration : 20 m/min
- Fast/Emergency Stop : 50 m/min

E.2 Basic Requirements System Control

E.2.1 Automation

A large number of automatic sequences and *models* will be implemented in the DVL2, e.g.:

1. automatic handling of *coils* and *strip* in the Entry and Exit of the line

2. automatic *setpoint calculation* for tension and the furnace control loops
3. automatic setpoint calculation for the airknife
4. automatic slow down based on measured overthickness of the strip

We require that the different automatic sequences and models/tables for the *setpoint generation* will be implemented in a uniform manner in the software. Please indicate the way you implement these functions in your system.

E.2.2 Man Machine Communication

All information exchange between the operators and the control system will be handled by this function. The following devices are to be considered:

1. color VDU's and keyboards
2. operator stations with pushbuttons and signallamps

Because of the complexity of the control system a large amount of the information exchange between the operator and the system will take place through VDU and keyboards. We require that similar functions on different locations will have the same MMC principles. Please indicate the possibilities of your system.

E.2.3 Alarm System

Because of the level of automation and the size/complexity of the control system and the processes, an extensive alarm and diagnosis system must be incorporated in the control system. Since the high number of expected alarms and messages, great emphasis must be put on the selective presentation of these alarms. Please present the possibilities of your system.

E.2.4 *Data Acquisition*

This function must be able to handle two kinds of data acquisition, *time related* and *strip related* . For the time related data acquisition we are thinking of constantly storing a few hundred signals for the past X hours with a resolution of 1 second to a couple of minutes. For the strip related acquisition we are thinking of storing about 50 signals every 25m of strip.

E.2.5 Interface with Production Control System

The control system for the DVL2 must be linked to the HO-Production control system (to be built by HO), by means of our *HDN (Hoogovens Data Network) communications link* , which is an upgrade of the Ethernet/DECnet communication link. *Coil and order related data* will be interchanged by means of HDN between the control system and the Production control system.

E.2.6 I/O System

We are considering the possibilities of the use of remote I/O. Please indicate what the possibilities of your system are in this area.